Pixelating Vector Art

by

Tiffany C. Inglis

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, 2014

I hereby declare that I am the sole author of this thesis, except where noted. 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.

Abstract

Pixel art is a popular style of digital art often found in video games. It is typically characterized by its low resolution and use of limited colour palettes. Pixel art is created manually with little automation because it requires attention to pixel-level details. Working with individual pixels is a challenging and abstract task, whereas manipulating higher-level objects in vector graphics is much more intuitive. However, it is difficult to bridge this gap because although many rasterization algorithms exist, they are not well-suited for the particular needs of pixel artists, particularly at low resolutions. In this thesis, we introduce a class of rasterization algorithms called pixelation that is tailored to pixel art needs. We describe how our algorithm suppresses artifacts when pixelating vector paths and preserves shape-level features when pixelating geometric primitives. We also developed methods inspired by pixel art for drawing lines and angles more effectively at low resolutions. We compared our results to rasterization algorithms, rasterizers used in commercial software, and human subjects—both amateurs and pixel artists. Through formal analyses of our user study studies and a close collaboration with professional pixel artists, we showed that, in general, our pixelation algorithms produce more visually appealing results than naïve rasterization algorithms do.

Acknowledgements

I would like to thank

- Craig Kaplan, my supervisor, for his guidance and support in graduate school, for encouraging me to pursue many interesting research topics over the years, and for introducing me to the world of mathematical art.
- Daniel Vogel, my co-author on a conference paper [36], for always being there when I need to discuss research ideas with someone and for providing great advice.
- other professors and fellow graduate students in the Computer Graphics Lab for a friendly work environment that promotes creativity and collaboration.
- Sven Ruthner, for spending long hours drawing pixel art for our research and offering insightful feedback that allowed us to gain a better understanding of how pixel artists work.
- eBoy artists, for always being quick to respond to any request and for providing encouraging feedback on our pixel art research.
- various pixel artists for their beautiful artworks featured in this thesis and/or for their tutorials that served as useful references.
- my undergraduate research assistants, Grace Chan, Xinyuan Fan and Tara Munikar, for their contribution to this work and in related projects that will provide as groundwork for future research.
- the Statistical Consulting Service for helping us with statistical analyses of user study results.
- Dr. Paul Pamboukian for giving me an emergency root canal the day before my defense.

Dedication

I would like to dedicate this work to my husband, Stephen. In addition to being a great partner, he has always taken a keen interest in my work, which has been a tremendous source of motivation and encouragement. I would also like to thank my parents for their endless support so that I may pursue a career I love.

Table of Contents

List of Tables xi

List of Figures xii

1 Introduction 1

1.1 Research Objectives and Contributions 5

1.2 Thesis Outline 6

2 Background 9

2.1 Styles of Pixel Art 9

2.1.1 Isometric Pixel Art 13

2.2 Pixel Art Workflow 16

2.2.1 Line Art 17

2.2.2 Antialiasing 20

2.3 Related Research in Computer Graphics 23

2.3.1 Rasterization 24

2.3.2 Font Rasterization 25

2.3.3 Pixel Art Research 27

2.4 Summary 29

3 Pixelating Paths **30**
3.1 Challenges in Rasterizing Line Art . . . 30
3.1.1 Dropouts . . . 31
3.1.2 L-shaped Corners and Pixel Clusters . . . 32
3.1.3 Jaggies . . . 33
3.2 Grid Alignment and Blips . . . 38
3.3 Algorithmic Approach . . . 40
3.3.1 Path Splitting . . . 41
3.3.2 Preprocessing: Shifting for Grid Alignment . . . 42
3.3.3 Naïve Rasterization . . . 43
3.3.4 Postprocessing: Partial Sorting for Jaggie Removal . . . 45
3.4 User Study . . . 49
3.4.1 Drawing Phase . . . 49
3.4.2 Rating Phase . . . 51
3.4.3 Results and Analysis . . . 54
3.5 Summary . . . 60

4 Pixelating Shapes **62**
4.1 Ellipse Test . . . 62
4.2 Algorithmic Approach . . . 65
4.2.1 Bounding Box Corner Alignment . . . 65
4.2.2 Preserving Other Global Properties . . . 66
4.3 Evaluation . . . 69
4.3.1 Ellipse Test for Superpixelator . . . 69
4.3.2 Qualitative Evaluation Set-up . . . 70
4.3.3 Qualitative Results Inspection . . . 70
4.3.4 Artist Feedback . . . 73
4.4 Summary . . . 74

5 Manual Antialiasing **76**
5.1 Antialiasing Algorithms . . . 77
5.2 Algorithmic Approach . . . 77
5.3 Qualitative Evaluation . . . 80
5.4 Qualitative Results Inspection . . . 80
5.5 Summary . . . 84

6 Drawing Straight Lines **85**
6.1 Aliased vs. Antialiased Lines . . . 87
6.2 Euclidean Line-Drawing Algorithm - Aliased . . . 89
6.2.1 Binary Representation of a Pixelated Line . . . 92
6.2.2 Euclidean Algorithm . . . 94
6.2.3 Algorithm Details . . . 94
6.2.4 Error Bound . . . 96
6.3 Euclidean Line-Drawing Algorithm - Antialiased . . . 98
6.3.1 Drawing Antialiased Units by Wu's Algorithm . . . 99
6.3.2 Determining the Sizes of Antialiased Units . . . 101
6.3.3 Antialiasing Parameters . . . 103
6.3.4 Antialiasing with a Custom Palette . . . 104
6.4 Quality of Pixelated Lines . . . 108
6.5 Run-time Analysis . . . 113
6.5.1 Using the Binary Representation . . . 115
6.5.2 Using the Compact Representation . . . 116
6.5.3 Comparison to Bresenham's Line Algorithm . . . 120

7 Drawing Angles **122**
7.1 Angles in Isometric Pixel Art . . . 124
7.2 Angle Survey . . . 126

7.3 Angle Classification . . . 131
7.4 Angle-Drawing Algorithm for Perfect Slopes . . . 131
7.4.1 Phase Shifting and Angle Widening . . . 131
7.4.2 Edge Offsetting . . . 133
7.4.3 Merging Truncated Spans . . . 134
7.4.4 Join Edges without Intersection . . . 136
7.4.5 Results and Evaluation . . . 138
7.5 Angle-Drawing Algorithm for Imperfect Slopes . . . 139
7.5.1 Line Slopes and Levels . . . 139
7.5.2 Algorithm Details . . . 140
7.6 Drawing Polygonal Paths . . . 142
7.7 Summary . . . 145

8 Conclusions and Future Work **147**
8.1 Superpixelator (without Antialiasing) . . . 147
8.2 Manual Antialiasing Algorithm . . . 150
8.3 Euclidean Line-Drawing Algorithm . . . 150
8.4 Angle-Drawing Algorithm . . . 151
8.5 Other Research Opportunities in Pixel Art . . . 152

APPENDICES **155**

A Partial Sorting Parameters **156**

B User Study Statistical Analysis **158**

C User Study Questionnaire **160**

D User Study Demographic Information **162**

E ELDA-A Error Bound **165**

F Angle-Drawing Algorithm Results **172**

References **179**

List of Tables

3.1 p-values of pairwise group comparisons for visual appeal 58
3.2 p-values of pairwise group comparisons for similarity 58
3.3 Rankings of visual appeal and similarity 58

6.1 Recursion variables 95
6.2 Values of convergents 99

E.1 Recursion variables for error bound calculation 168
E.2 Values from the Fibonacci and a Fibonacci-like sequence 170
E.3 Error bound values 171

List of Figures

1.1 Pixel art in games 2
1.2 Pixel art outside of games 3
1.3 Pixel art landscape 4
1.4 Rasterized vector art vs. pixel art 5
1.5 Thesis summary 7

2.1 Pixel art from Castlevania 10
2.2 Triangular representation of the pixel art space 11
2.3 Realistic pixel art 12
2.4 Abstract pixel art 13
2.5 Symbolic pixel art 14
2.6 Pixel-level details 14
2.7 Isometric pixel art 15
2.8 Isometric lines and angles 15
2.9 Two ways to start drawing pixel art 16
2.10 Yu's tutorial on a typical pixel art workflow 17
2.11 Parabolas of different resolutions 18
2.12 Pixel spans of various lengths and orientations 18
2.13 Perfect vs. imperfect lines 19
2.14 Jaggies in lines and curves 19
2.15 Different ways to draw paths and filled shapes 21

2.16 Pixel patterns in manual antialiasing . . . 22
2.17 Different ways to draw a filled circle . . . 22
2.18 Manual antialiasing with non-greyscale colours . . . 23
2.19 Variations in manual antialiasing . . . 24
2.20 Different types of font hints . . . 26
2.21 Font rasterization . . . 26
2.22 Pixel art scaling algorithms . . . 28
2.23 Algorithms for creating pixel art . . . 29

3.1 Pixel line art . . . 31
3.2 Dropouts . . . 32
3.3 Definition of L-shaped corners . . . 33
3.4 L-shaped corner removal . . . 33
3.5 Slope sequence representation of a pixelated path . . . 34
3.6 Splitting a path to resolve ambiguity in slope sequence . . . 35
3.7 Monotonicity and slope-monotonicity . . . 35
3.8 Smooth vs. jagged paths . . . 36
3.9 Identifying jaggies in a path . . . 37
3.10 Exceptions involving jaggies . . . 38
3.11 Removing blips via grid alignment . . . 39
3.12 Grid-aligning a circle . . . 40
3.13 Points at which to split a path . . . 41
3.14 Grid alignment steps . . . 42
3.15 Steps in inflection point shifting . . . 43
3.16 Steps in Bresenham's line algorithm . . . 45
3.17 Subdivision and removal of redundant pixels . . . 46
3.18 Examples of naïve path sorting . . . 47
3.19 A comparison of pixelation results using different cost functions . . . 48

3.20 The neighbourhood of a pixelated path 49
3.21 A screenshot from the Drawing Phase of the user study 50
3.22 Images used in the user study 51
3.23 Pixel art results from Pixelator, Illustrator, and Photoshop 52
3.24 Screenshots from the Rating Phase of the user study 53
3.25 Examples of hand-drawn pixel art from the user study 54
3.26 The averages of the images collected in the Drawing Phase 55
3.27 Visual appeal preference histogram 56
3.28 Similarity preference histogram 56
3.29 Hand-drawn images with the lowest ratings 59
3.30 Hand-drawn images with the highest ratings 60
3.31 Rasterized vs. hand-drawn lines and angles 61

4.1 Ellipses used for the ellipse test 63
4.2 Ellipse test for symmetry 63
4.3 Artifacts to look for in the ellipse test 64
4.4 Ellipse test for blips, L-shaped corners, and dropouts 64
4.5 Preserving elliptical symmetry by bounding box adjustment 66
4.6 Shape-level rasterization artifacts 66
4.7 Before and after bounding box corner alignment 67
4.8 Ellipse test for Superpixelator 69
4.9 Vector shapes rasterized without antialiasing 71
4.10 Comparing path smoothness among rasterized shapes 72
4.11 Comparing shape symmetry among rasterized shapes 72
4.12 Comparing sharp angle preservation among rasterized shapes 73
4.13 A screenshot of Superpixelator 74
4.14 Pixel pattern comparison in aliased lines 75

5.1 No antialiasing, manual antialiasing, and automatic antialiasing 77
5.2 Steps in our manual antialiasing algorithm 79
5.3 Vector shapes rasterized with antialiasing 81
5.4 Comparing blurriness among antialiased shapes 82
5.5 Comparing apparent path thickness among antialiased shapes 83
5.6 The original and the darkened version of the pixel artist's results 83
5.7 Identifying repeating units of pixel patterns in the antialiased results . . . 84

6.1 Pixel regularity comparison 86
6.2 Isometric truck and skeletal structure 87
6.3 Aliased units and antialiased units 88
6.4 Bresenham's attempts to draw perfect lines 89
6.5 Variation in antialiased units used in an antialiased line 90
6.6 Symmetric arrangement of aliased units to form longer segments 91
6.7 Steps in ELDA-A 92
6.8 The lengths of unitss do not affect their arrangement 93
6.9 Initial set-up for ELDA-A 95
6.10 How to reduce computation in the recursion 97
6.11 The recursive construction of intermediate segments in ELDA-A 97
6.12 Steps in Wu's antialiasing algorithm 100
6.13 Minimally antialiased line 102
6.14 Moderately antialiased line 102
6.15 ELDA-AA parameters 105
6.16 Varying the apparent thickness with a 5-colour palette 106
6.17 Effect of different greyscale conversion on antialiasing 107
6.18 Results from manual antialiasing with non-greyscale palettes 108
6.19 Lines antialiased with different number of colours 108
6.20 Heat map of the quality of antialiased lines using ELDA-AA 110

6.21 Fixing the quality measure for aliased lines . . . 111
6.22 Heat map of the quality of aliased lines using ELDA-A . . . 112
6.23 Line snapping based on quality measures . . . 114
6.24 Line snapping applied to an isometric cube . . . 115
6.25 Run-time analysis of ELDA-A using binary representations . . . 117
6.26 Compact representation of a pixelated line . . . 118
6.27 Run-time analysis of ELDA-A using compact representations . . . 119
6.28 Run-time analysis of Bresenham's line-drawing algorithm . . . 120

7.1 Terminology for angles . . . 123
7.2 Artifacts in naïvely drawn angles . . . 123
7.3 Angle extraction from isometric pixel art . . . 124
7.4 Problems in angle extraction . . . 125
7.5 Extracted angles from isometric pixel art . . . 126
7.6 Basic angle rules found in eBoy's pixel art . . . 127
7.7 Angle survey question 1 . . . 127
7.8 Angle survey question 2 . . . 128
7.9 Angle survey question 3 . . . 128
7.10 Angle survey question 4 . . . 129
7.11 Angle survey question 5 . . . 129
7.12 Angle survey question 6 . . . 130
7.13 Angle survey question 7 . . . 130
7.14 Octants and quadrants . . . 132
7.15 Octant and quadrant range . . . 133
7.16 Angle widening . . . 134
7.17 Very narrow angles without offset . . . 135
7.18 Narrow angle offset direction . . . 135
7.19 Merged span lengths . . . 136

7.20 Forming merged spans from truncated spans . . . 137
7.21 Algorithm results for angles with quadrant ranges of 3 . . . 138
7.22 Levels of lines . . . 139
7.23 Pixel patterns in imperfect lines . . . 140
7.24 Very narrow angle with imperfect slopes . . . 141
7.25 Narrow angle with imperfect slopes . . . 141
7.26 Wide angles with imperfect slopes . . . 142
7.27 Drawing polygonal paths . . . 143
7.28 How edge lengths change in polygonal path drawings . . . 144
7.29 How to limit length changes in previously drawn edges . . . 145
7.30 Polygonal path drawn with good angles . . . 146

8.1 Pixel art reproduced by Superpixelator . . . 148
8.2 Preserving topology . . . 149
8.3 Multi-resolution abstraction . . . 149
8.4 Tapering at joints . . . 150
8.5 Palette arrangement in pixel art . . . 152
8.6 Dithering in pixel art . . . 153
8.7 Tilesets in pixel art . . . 154

D.1 Age distribution of user study participants . . . 162
D.2 Gender distribution of user study participants . . . 163
D.3 Artistic level among user study participants . . . 163
D.4 Familiarity with pixel art among user study participants . . . 164
D.5 Pixel art experience of user study participants . . . 164

E.1 The error of a pixelated approximation of a line . . . 166
E.2 The error introduced at each step of the algorithm . . . 167
E.3 Preliminary error bound . . . 168

F.1 Angles composed of level 1 edges drawn with our algorithm 173
F.2 Angles composed of level 1 edges drawn with the naïve algorithm 174
F.3 Angles composed of level 2 edges drawn with our algorithm 175
F.4 Angles composed of level 2 edges drawn with the naïve algorithm 176
F.5 Angles composed of level 3 edges drawn with our algorithm 177
F.6 Angles composed of level 3 edges drawn with the naïve algorithm 178

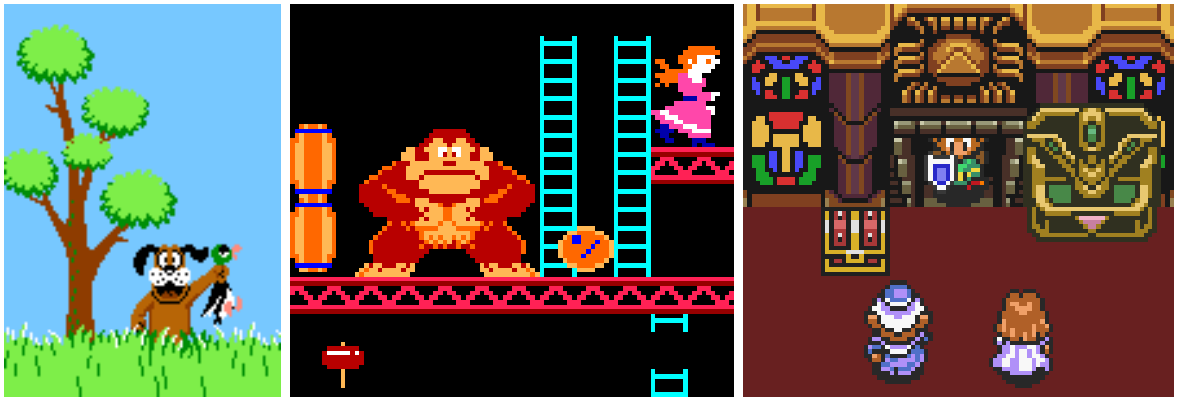
Chapter 1

Introduction

Pixel art is a style of digital art created and edited on the pixel level. It is most commonly associated with video games from the 1980s, because it was originally developed out of necessity when early 8-bit graphics hardware had limited resolutions and colour palettes. Traditional pixel art uses large blocky pixels and few colours to draw highly abstracted objects for which each pixel has significance. Combined with the colour limitation, creating pixel art was and remains a difficult task that can only be carried out by hand with careful planning and editing.

What we now think of as pixel art used to be just computer graphics. It was formerly a necessity but now it is a conscious choice. Many classic video games from the 1980s and early 1990s such as Duck Hunt, Donkey Kong, and The Legend of Zelda feature pixel art graphics (see Figures 1.1a, 1.1b, and 1.1c) that have become cultural icons due to their distinctive styles. The hardware limitations of the time did not hinder the progress of video game graphics; instead, they helped create a unique genre of art. Even after 1990, when video games began to use 3D graphics with higher resolutions, pixel art remained a popular choice for game art.

Nowadays, computers and game consoles offer graphics performance orders of magnitude beyond what was possible twenty years ago. Surprisingly, there is still a growing number of pixel art games, some designed to evoke nostalgia for older games, others exploring new art forms that depart from realism. Owlboy (see Figure 1.1d) recreates the vintage feel of retro pixel art games with intricately detailed scenes drawn using many well-chosen colour palettes. The Pokémon series (see Figure 1.1e), created for handheld game consoles, continues to use pixel art graphics, even though some 3D components have been added for the newer games. Superbrothers: Sword & Sworcery EP (see Figure 1.1f)



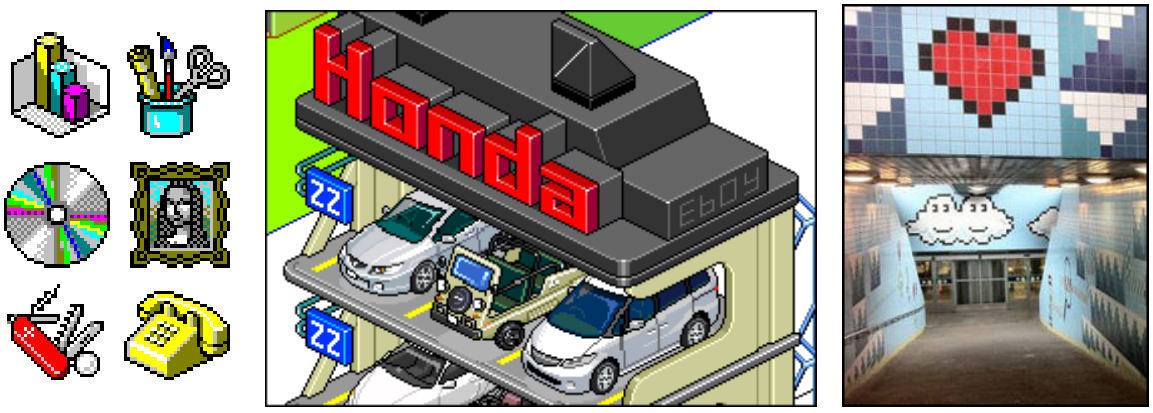
(a) (b) (c)

(d) (e) (f)

Figure 1.1: Examples of pixel art in games: (a) Duck Hunt (1984), (b) Donkey Kong (1981), (c) The Legend of Zelda: A Link to the Past (1991), (d) Owlboy (2013 for demo version), (e) Pokémon Black and White (2010), and (f) Superbrothers: Sword & Sworcery EP (2011).

combines minimalistic pixel art with higher-resolution graphics to create a hybrid art form. Minecraft (2011) extends pixel art to three dimensions, creating a world of voxels painted with pixel art textures.

Pixel art can also be found outside of games. Computer icons, especially those on older operating systems such as Windows 3.1, are designed as pixel art (see Figure 1.2a). Many well-known companies such as Honda, Sony, Yahoo! and Coca-Cola have also advertised through pixel art. Figure 1.2b shows the Honda Icon Museum, an award-winning interac-



(a) (b) (c)

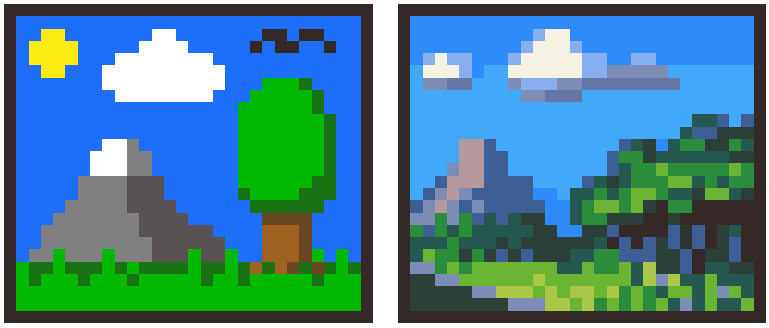
Figure 1.2: Examples of pixel art outside of games: (a) icons from Windows 3.1, (b) Honda advertisement by eBoy, and (c) Stockholm subway station decorated with tiled pixel art.

tive campaign designed by eBoy for Honda.[1] Pixel art is also a convenient form of artistic expression whenever the canvas has a grid-like structure. A Stockholm subway station has recently been refurbished with pixel art made out of tiles (see Figure 1.2c). Other art forms such as mosaic, cross-stitch, and beadwork sometimes use pixel art designs since they can be easily adapted.

Creating pixel art does not require any specialized software tools. It can be drawn using any raster image editor, even one as basic as Microsoft Paint.[2] Many pixel artists prefer to use tools such as GraphicsGale[3] or ProMotion[4] because they are specifically designed for pixel art and offer better support for working with low-resolution compositions, editing limited colour palettes, and creating animations. Regardless of what tools are used, it is important that the artist not rely on them too much. Pixel art is all about control, and to have all the pixels in exactly the right places, pixel artists do most of the work by hand, placing pixels one by one with little automation. Many pixel drawings are of sufficiently low resolution that every pixel must be positioned precisely to avoid changing what is being conveyed and, to achieve this, artists must rely on manual pixel-level adjustment.

Even though many pixel art compositions have low resolutions and use few colours, they can be richly expressive if the right pixels are used. Because pixel art is drawn pixel

[1] hello.eboy.com/eboy/2006/02/07/honda-icon-museum-wins-gold-prize/
[2] http://windows.microsoft.com/en-us/windows7/products/features/paint
[3] www.humanbalance.net/gale/us/
[4] www.cosmigo.com/promotion/



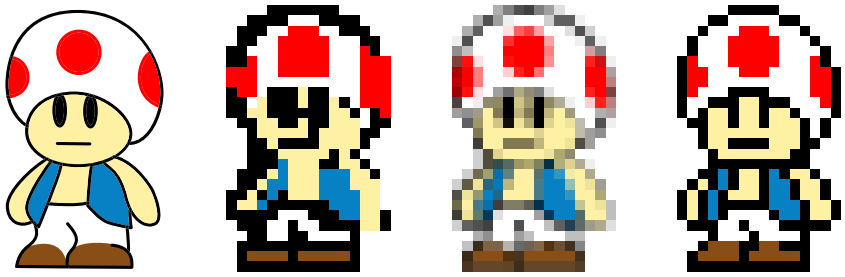
(a) (b)

Figure 1.3: Two pixel art landscapes by Arne Niklas Jansson. Used with permission.

by pixel, it can be difficult to understand each pixel's contribution to the whole. For example, Figure 1.3 shows two pixel art landscapes.[5] In Figure 1.3a, it is easy to tell, for example, that the blue pixels represent the sky, the black pixels suggest a flying bird, and that the lighter and darker green pixels are used for lighting and shading. However, this composition is overly simplistic and does not provide much visual interest. The image in Figure 1.3b has a much more complex structure; in this image, it is more difficult to determine what individual pixels represent, but together they create a composition that contains much more depth. A good pixel artist needs to understand how to work with pixels at such a low level without losing sight of the bigger picture. This skill can take years to master even for someone with artistic training.

On the other hand, digital artists know how to work quite expressively with vector illustration software, where they can operate at a higher level, dealing in form and composition. If pixel art could be designed using standard vector illustration tools, it would be much easier to edit and become much more accessible to amateurs. Of course there are many software tools that can rasterize vector art, but the resulting raster image is often of poor quality by pixel art standards. For example, Figure 1.4a shows a vector image. Rasterized without antialiasing (Figure 1.4b), the outlines are broken and details are hard to read. Rasterizing with antialiasing (Figure 1.4c), it looks too blurry and again details are lost. These results should come as no surprise, since rasterization algorithms simply are not designed to produce convincing results at such low resolutions. In contrast, hand-drawn pixel art designed pixel by pixel (Figure 1.4d) contains just the right amount of detail, appropriately abstracted for the intended resolution. Some vector graphics editors incorporate a few tools to facilitate rasterization (for example, Adobe Illustrator's Snap to

[5] http://androidarts.com/pixtut/pixelart.htm



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

Figure 1.4: (a) Vector art, (b) vector art rasterized without antialiasing, (c) vector art rasterized with antialiasing, and (d) hand-drawn pixel art. The rasterization is done using Adobe Illustrator.

Pixel function) but they offer only minor improvements and the quality of the results is nowhere near that of hand-drawn pixel art.

Fundamentally, what we are looking for is smarter rasterization algorithms that can properly convert vector images into pixel art drawings. Such algorithms would let artists operate at a higher level while relegating the task of making complex pixel-level decisions to the computer. A vector image contains various components, including paths (i.e., lines, curves, and shape primitives), text, or colour gradients. Our rasterization algorithm, called *pixelation*, focuses specifically on rasterizing vector paths in a way that meets pixel art standards. We achieve this goal by studying methods used by artists to assess the quality of pixel art drawings, and then using these criteria to design our algorithms.

1.1 Research Objectives and Contributions

Our main research objective can be summarized as follows::

Understand how pixel artists cope with resolution constraints, and use this knowledge to create new rasterization algorithms tailored to pixel art.

This goal can be divided into several subproblems. In particular, we wish to address the following research questions:

1. What are the main challenges concerning the creation of pixel art?

2. What are some methods of assessing the quality of pixel art?

3. How does hand-drawn pixel art differ from images created by rasterization algorithms?

4. How do we describe algorithmically what pixel artists do by hand?

To answer these problems, we conducted a survey of pixel art techniques by studying various online tutorials to understand the basic workflow and some of the challenges pixel artists face. From these tutorials, we also learned what features to look for to differentiate good pixel art from bad pixel art. Then we used these criteria to evaluate rasterization algorithms, pinpointed areas that need improvement, and designed our algorithms to overcome these issues.

Our main contributions are as follows:

1. We provide a summary of different styles of pixel art and various techniques used to draw pixel art.

2. We present precise mathematical definitions for various artifacts in pixel art that we want to avoid.

3. We collected large datasets of hand-drawn pixel art images and rasterized images [33, 34] that can be useful for future research.

4. We developed three pixelation algorithms: (1) Superpixelator pixelates generic Bézier splines and shapes primitives, (2) the Euclidean Line-Draw Algorithms pixelate straight lines, and (3) the Angle-Drawing Algorithm pixelates angles formed by line segments. Compared to traditional rasterization algorithms, our pixelation algorithms create more visually appealing results, especially at low resolutions.

To our knowledge, there has been no work done in converting vector graphics to pixel art. We believe that our work can not only benefit pixel artists and game developers, but also offer new ideas in the field of rasterization inspired by pixel art practices.

1.2 Thesis Outline

The remainder of the thesis is organized as summarized in Figure 1.5.

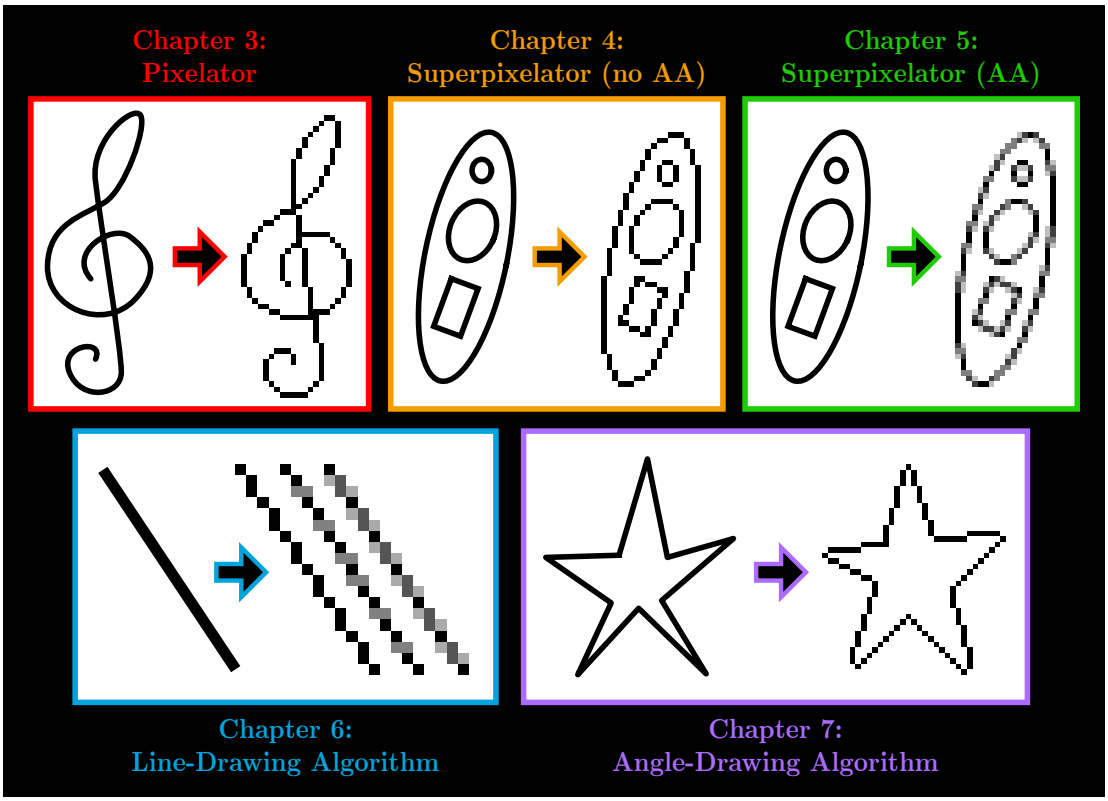


Figure 1.5: A summary of the algorithms described in each chapter. *AA* stands for antialiasing.

Chapter 2 provides background knowledge for pixel art and rasterization. We describe various styles of pixel art, components within a composition, and techniques used by pixel artists. We also discuss related research in computer graphics, including rasterization, antialiasing, and font rasterization.

In Chapter 3, we discuss the challenges in pixelating vector paths, focusing on various types of artifacts to avoid. Then we describe a pixelation algorithm called Pixelator that suppresses each type of artifact. The chapter concludes with a user study to evaluate our results.

Chapter 4 is an extension of Chapter 3 in which we discuss additional concerns involving pixelated shape primitives. We describe several artifacts related to shape rasterization and how to avoid them in our improved algorithm called Superpixelator. For evaluation, we collaborated with professional pixel artists.

In Chapter 5, we introduce a style of antialiasing used in pixel art called manual antialiasing. First we compare it to typical antialiasing algorithms. Then we describe our algorithm that mimics the appearance of manual antialiasing. The evaluation method is the same as that of Chapter 4.

Chapter 6 focuses on a line-drawing algorithm based on the Euclidean algorithm, whose main goal is to produce regular pixel patterns found in hand-drawn pixel art. We describe some interesting number-theoretic properties of the algorithm and analyze how its running time scales with different inputs.

In Chapter 7, we describe an angle-drawing algorithm inspired by isometric pixel art. The algorithm handles different types of angles differently, and is extended to draw polygonal paths. The results were evaluated by eBoy, from whom we drew inspiration.

Chapter 8 summarizes our findings and describes the limitations of our algorithms. For future work, we discuss how our work may be improved as well as some challenges that lie in other areas of pixel art.

Chapter 2

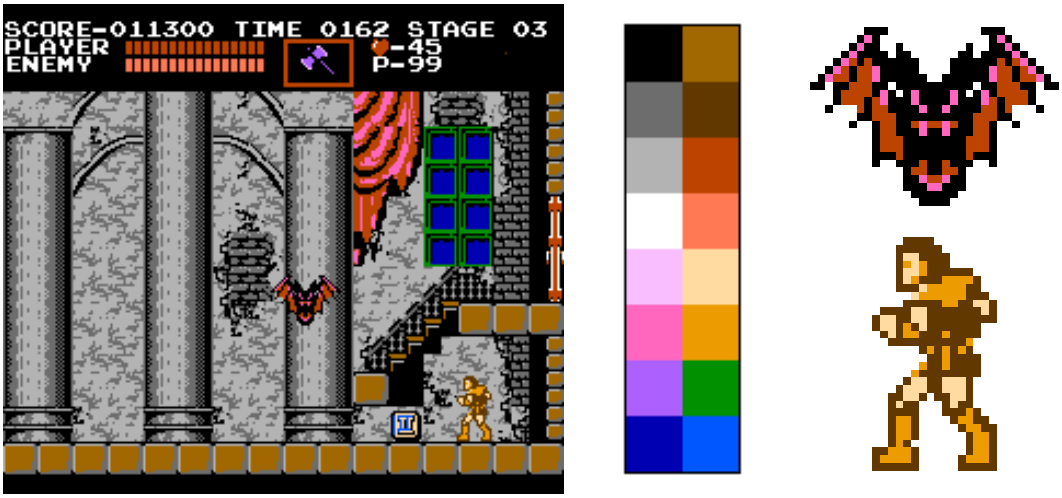
Background

In this chapter, we give an overview of pixel art both as an artistic medium and as an area of research within computer graphics. First, we try to gain a deeper understanding of what pixel art means by exploring its stylistic range. Then we follow several commonly referenced pixel art tutorials to establish a typical workflow, including steps such as outlining, colouring, and shading. We describe the goal of each step, consider possible issues, and demonstrate its application with various examples.

After studying theories developed by pixel artists, we discuss related research in computer graphics. We begin by describing several rasterization and antialiasing algorithms, and how they perform under pixel art constraints. Then we cover font rasterization in detail since it emphasizes low-resolution rasterization. Finally we discuss algorithms developed specifically for pixel art, including vectorization and upsampling methods.

2.1 Styles of Pixel Art

Most people associate pixel art with the type of graphics used in retro arcade games, with low resolutions and bright colours. Figure 2.1a shows an example from Castlevania, a game released in 1986 that uses pixel art. For this particular scene, the artist was restricted to using only 16 vibrant colours (see Figure 2.1b). For each character (see Figure 2.1c), there is a size restriction of no more than 30 pixels tall as well as a palette restriction of no more than three colours. These constraints were difficult to work with but pixel artists had no choice but to respect the limitations of the hardware of the day.



(a) (b) (c)

Figure 2.1: Pixel art from Castlevania (1986): (a) screenshot, (b) 16-colour palette, and (c) two characters.

As technology advanced, the highly restrictive resolution and palette constraints were gradually lifted. Artists can now afford to use more pixels and more colours; as a result, many new styles of pixel art have been developed. Some artists artificially enlarge their pixels to mimic the retro style, while others work at screen resolution and strive for realism. Pixel artists still choose to work with limited palettes—possibly for historical reasons—but usually they can choose their own colours.

Pixel art covers such a wide spectrum of styles that it is often difficult to discern if a composition is in fact pixel art. Inspired by Scott McCloud's theory of comics [46], pixel artist Helm uses a triangle (see Figure 2.2) to visualize the space of pixel art.[1] The three vertices represent three stylistic extremes: realism, abstraction, and symbolism. We can describe the style of a pixel art composition in terms of its position in this triangle.

Realistic pixel art images, in the extreme case, look like photographs drawn with perfect pixel precision. Two examples of realistic pixel art are shown in Figures 2.3 along with their reference paintings. To see why these pixel art images are difficult to reproduce by automatic means, we scaled each painting down to the same height as its pixel art counterpart while preserving the aspect ratio. Then we reduced the number of colours

[1] http://www.pixel.schlet.net

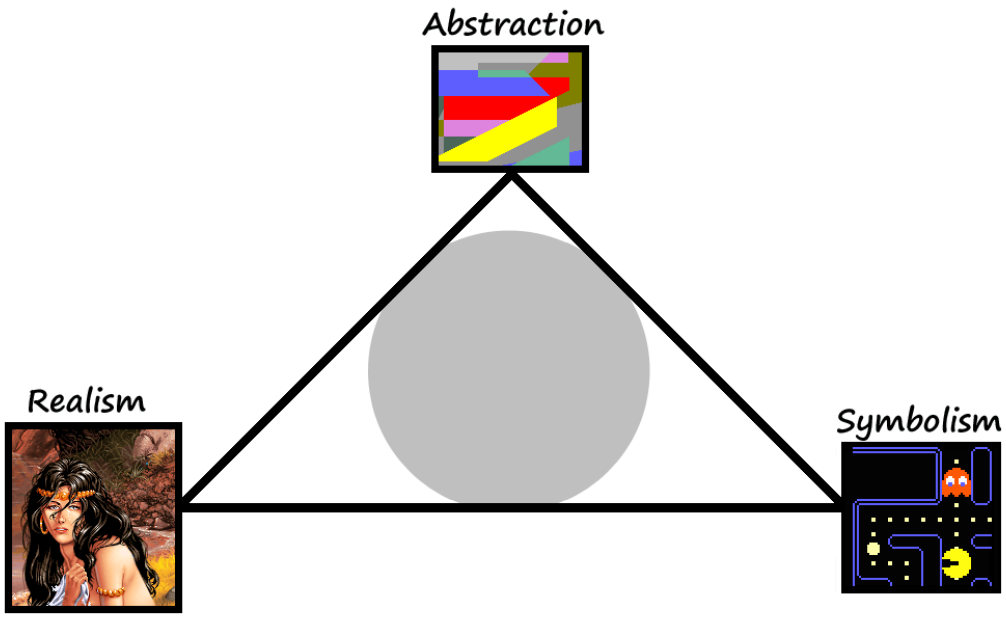


Figure 2.2: The space of pixel art represented as a triangle with three extreme styles: realism, abstraction, and symbolism. Used with permission.

using colour quantization, both with and without dithering. For dithering, we used the Floyd-Steinberg algorithm [49].

The resized and colour quantized paintings are shown in Figures 2.3c, 2.3d, 2.3g and 2.3h. In both cases, the pixel art images are much more cleanly drawn and detailed due to the pixel artists' attention to detail and their ability to emphasize important features in the compositions, even though the pixel art image in Figure 2.3e uses a very different colour palette compared to the original painting. In contrast, the dithered paintings are much more noisy and the undithered paintings contain patches of colours that do not suggest form as meaningfully as the pixel art images do.

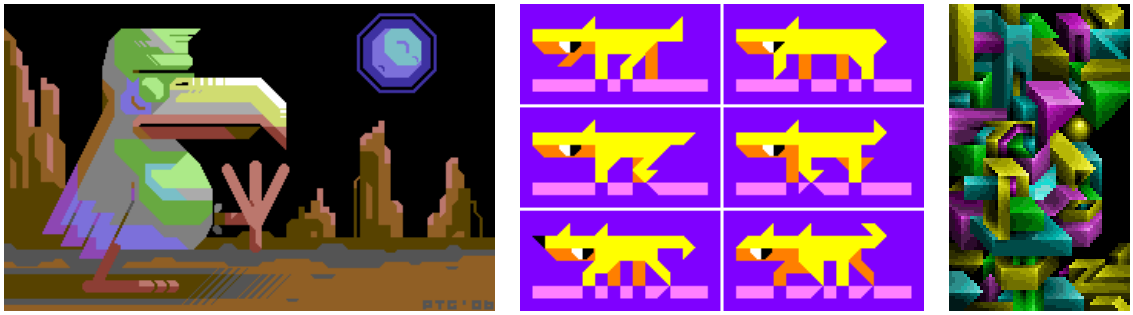
Abstract pixel art is more experimental and pushes the limit of what it means to be pixel art. Usually, the pixel size is a constraint we try to work around to create some artwork we have mind. An abstractionist, however, molds their artwork around this constraint to make the physical pixel limitation less apparent.

Abstract pixel art compositions often contain many geometric shapes, and emphasize horizontal and vertical lines because they can be drawn with perfect precision even at low resolutions. Lines whose slopes are integers or integer reciprocals are also used frequently

Figure 2.3: Realistic pixel art: (a) *!* by Cure (100×100 pixels with 24 colours), (b) its reference painting, and (c, d) the painting downscaled and colour quantized with and without dithering; (e) *Wishbone* by Helm, (cropped to 90×132 pixels with 16 colours), (f) its reference painting by Nikolaos Gyzis, and (g, h) the painting downscaled and coloured quantized with and without dithering. Used with permission.

in abstract pixel art. We call these slopes *perfect slopes* and lines with these slopes *perfect lines*; in Section 2.2.1, we will describe why these slopes are optimal for pixel art. Figure 2.4 shows three abstract pixel art compositions that are highly geometric and contain both 2D and 3D elements drawn using perfect lines.

Since they usually come from retro video games, most well-known pixel art icons can be considered symbolic with varying degrees of realism. Symbolic pixel artists celebrate the use of pixels rather than trying to hide it. In fact, the pixels are often enlarged to make them more visible. *Sprites* are good examples of symbolic pixel art; they are bitmap images



(a) (b) (c)

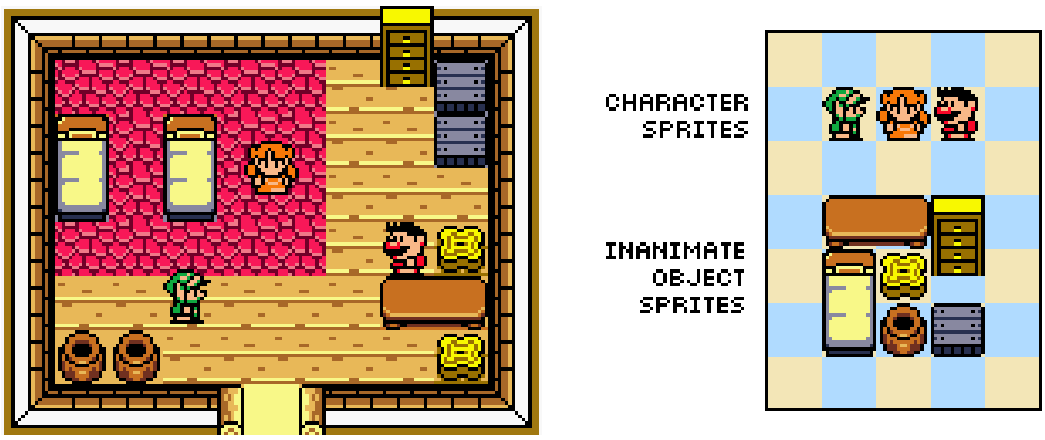
Figure 2.4: Abstract pixel art: (a) *Yus Chunk* by Sven Ruthner, (b) *Sharpdog* by tocky, and (c) *Portrait de Jeanne* by Graindolium. Used with permission.

that are integrated into a larger scene, most commonly used to represent characters or other foreground objects in games with static backgrounds. Figure 2.5a shows a screenshot from The Legend of Zelda: Link's Awakening. To fit as much information as possible onto a 160×144 Game Boy screen, the game uses many low-resolution sprites (see Figure 2.5b). Characters are given large heads to accentuate facial features and inanimate objects are often drawn at unrealistic relative sizes. Figure 2.5b shows that there is an underlying grid structure on which these objects are arranged. 8×8 grid sizes are commonly used; in this case, all the sprites either fit into one or two tiles.

Symbolic pixel art relies heavily on *pixel-level details*, which are details small enough that single-pixel alterations may significantly change their meaning. For example, Figure 2.6 shows several slight modifications to a sprite such as adding, removing, or shifting a pixel. Below each sprite, we show our vector interpretation of it—created by hand—to demonstrate how subtle changes can significantly change facial expressions, poses, and patterns.

2.1.1 Isometric Pixel Art

Isometric pixel art is a particular style of pixel art that is used in many video games. The idea is to use isometric projection to create a 3D scene that can be drawn optimally at low resolutions. When projected isometrically, a square tiling becomes the isometric grid shown in Figure 2.7a. This particular projection is most commonly used because it allows the coordinate axes to be drawn as perfect lines.



(a) (b)

Figure 2.5: Symbolic pixel art: (a) a screenshot from The Legend of Zelda: Link's Awakening (1993) and (b) sprites from the scene.

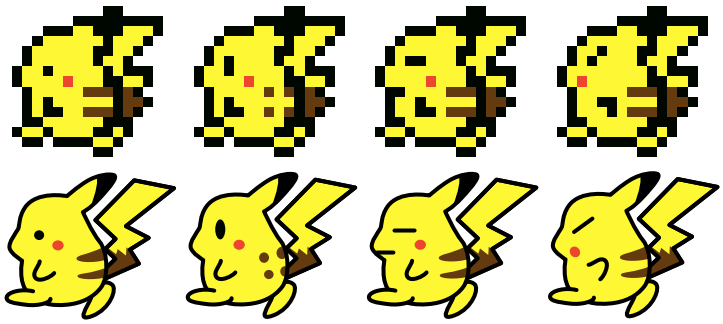
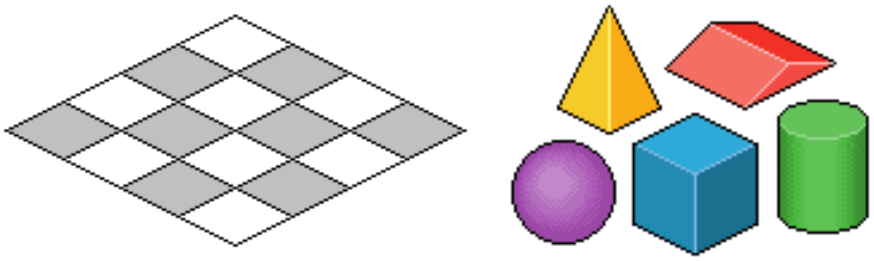


Figure 2.6: Small changes in pixel-level details can significantly change their meaning.

Using isometric projection, we can draw many 3D shapes (see Figure 2.7b) using perfect lines. These basic building blocks can be combined to form more complex 3D objects as shown. Figure 2.8a shows an example of an isometric pixel art scene. Notice that it has elements from both realism and symbolism: it depicts a realistic casino scene and it contains symbolic representations of real-life objects such as slot machines and people.

Isometric pixel artists tend to use a lot of straight lines in their artwork. Figure 2.8b contains the outlines of a slot machine extracted from Figure 2.8a. Notice that the artists have deliberately chosen only perfect lines. These lines are also carefully placed to ensure that they create nice-looking angles. In contrast, the angles in Figure 2.8c may be judged



(a) (b)

Figure 2.7: (a) The typical isometric grid and (b) isometric 3D shapes.

(a) (b) (c)

Figure 2.8: (a) A scene from eBoy's isometric pixel art (2013), (b) an outline of the slot machine from the scene, and (c) the slot machine drawn drawn with bad angles. Used with permission.

as less attractive. In Chapter 7, we describe more precisely how to determine the aesthetic quality of pixelated angles.

For our research, we focus mainly on the realistic and symbolic styles. We look at how to represent generic shapes in pixel art, which is important for the realistic style since we do not want to be restricted by what we can draw. We also consider pixel-level problems that occur at low resolutions, where symbolic representations become increasingly necessary. Finally, we focus on how to represent geometric objects such as lines and angles optimally, which are important for isometric pixel art.

(a) (b)

Figure 2.9: In Cure's tutorial, he describes two ways to begin drawing pixel art: (a) with the line art, or (b) with colour blocks. Used with permission.

2.2 Pixel Art Workflow

There are many online tutorials that give detailed instructions on how to draw pixel art. Most of them teach artistic techniques rather than how to use existing software tools since pixel artists are expected to do most of the work instead of relying on technology. A well-known tutorial by Cure[2] describes two common methods for starting a composition. The line art method, as shown in Figure 2.9a, outlines the image before filling in colours and other details. The colour-blocking method, as shown in Figure 2.9b, blocks in all the major forms using a large brush, then gradually refines the image down to pixel-level details.

While both methods are valid, the line art method seems more common as it is referred to more often in pixel art tutorials. Derek Yu's tutorial[3] breaks down the line art method into multiple steps. As shown in Figure 2.10, we start with the rough line art, clean it up, apply colours with shading and highlighting, add antialiasing and dithering, and then finally fill in details to complete the piece. For each step, Yu provides additional guidelines such as how to draw smooth-looking outlines and how to choose a good colour palette.

For the purposes of our research, we are mostly interested in how to draw line art and how to apply antialiasing as a pixel artist would. We will study both pixel art tutorials and related works in computer graphics in order to define our goals more precisely.

[2] www.pixeljoint.com/forum/forum_posts.asp?TID=11299

[3] makegames.tumblr.com/post/42648699708/pixel-art-tutorial

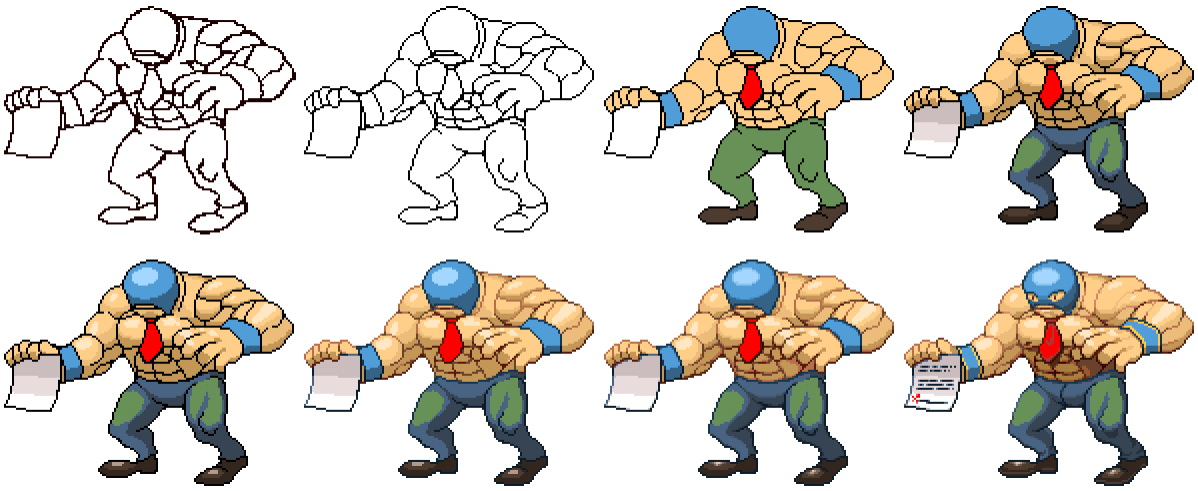


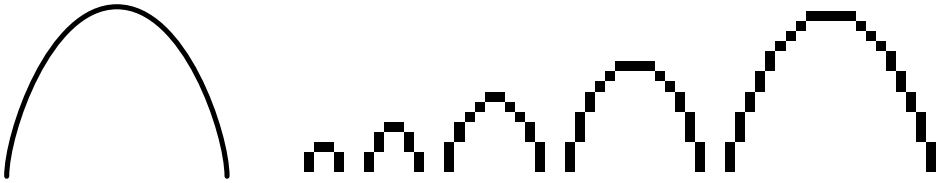
Figure 2.10: Yu's tutorial describe a typical pixel art workflow step by step: (1) rough outlining, (2) cleaning up to create line art, (3) flat colouring, (4) shading, (5) highlighting, (6) edge highlighting, (7) antialiasing, and (8) adding details. Used with permission.

2.2.1 Line Art

Line art refers to the outlines in an image. It consists of *paths*, which are either straight lines or curves. In pixel art, the line art is typically composed of one-pixel-thick paths. We can think of a pixelated path as an approximation of a vector path at a particular resolution. For example, Figure 2.11b shows several pixelated parabolas that approximate the vector parabola in Figure 2.11a at different resolutions. As the resolution decreases, it becomes increasingly difficult to approximate a vector path accurately. At low resolutions, typical rasterization algorithms will produce results that are unacceptable by pixel art standards since they contain many artifacts. We will start by describing some properties of pixelated lines and curves, introduce some relevant terminology, and define the types of artifacts to avoid.

In pixel art, straight lines are composed of fundamental units called *pixel spans*, or simply *spans*. The orientation of a span is either horizontal or vertical, and the length of a span is the number of pixels it contains. Figure 2.12 shows horizontal and vertical spans up to length 5.

A horizontal or vertical line is simply a long horizontal or vertical span of the same length, as shown in white in Figure 2.13. As for diagonal (i.e., non-horizontal and non-vertical) lines, they are created by joining spans at diagonally opposite corners. Some lines



(a) (b)

Figure 2.11: (a) A vector parabola and (b) parabolas drawn by hand at different resolutions.

Span Type	Length 1	Length 2	Length 3	Length 4	Length 5
Horizontal					
Vertical					

Figure 2.12: Pixel spans of various lengths and orientations

require only one type of span to draw—we call these *perfect lines*. Lines that require more than one type of span to draw are called *imperfect lines*. Figure 2.13 shows examples of both types of lines.

As shown in Figures 2.7 and 2.8, pixel artists—particular those that practice the isometric style—prefer perfect lines over imperfect lines. We believe this preference stems from the perceived straightness of the line. When we look at a sequence of spans joined corner-to-corner, we interpret it as a polygonal path formed by joining these corners. For a perfect line, this polygonal path is a straight line (see Figure 2.14a), whereas for an imperfect line, this path is jagged (see Figure 2.14b).

The *slope* of a pixelated line is defined as its height in pixels divided by its width in pixels. Whether a line is perfect or imperfect depends on its slope. Perfect lines have perfect slopes that are either integers or integer reciprocals. Figure 2.14a shows an example of a perfect line with slope 2. Imperfect lines have imperfect slopes, and they appear jagged due the transition between different types of spans. The pixel corner at which two different types of spans join is called a *jaggie*. Figure 2.14b shows an example of an imperfect

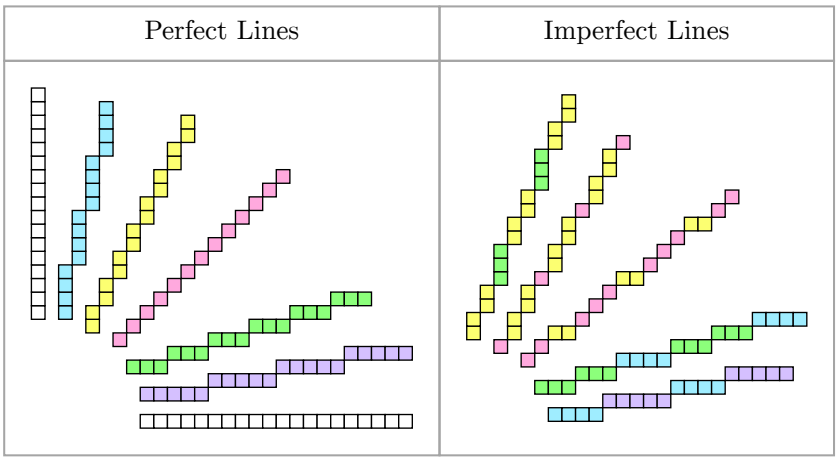


Figure 2.13: Perfect lines are composed of only one type of span, whereas imperfect lines consist of multiple types. Perfect lines have integer or integer reciprocal slopes.

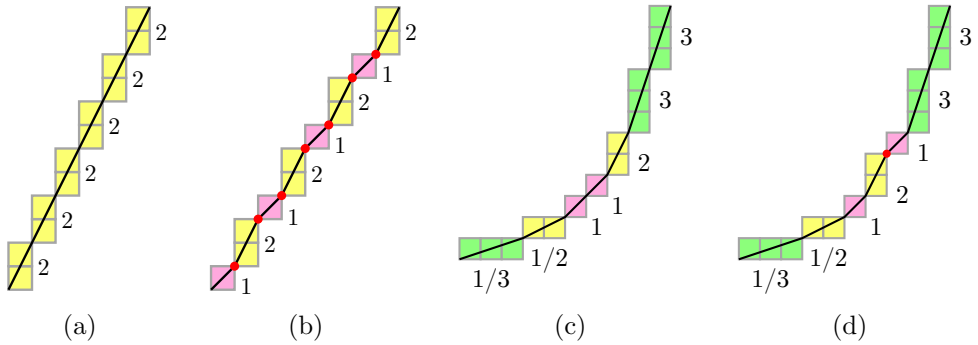


Figure 2.14: (a) A perfect line of slope 2, (b) an imperfect line of slope $\frac{3}{2}$, (c) a curve without jaggies, and (d) a curve with one jaggie. In each case, the polygonal path approximated by the pixelated line or curve is shown in black and the jaggies are shown in red. The numbers indicate the slopes of the line segments represented by the spans.

line with slope $\frac{3}{2}$ containing jaggies between its length-1 and length-2 spans. Jaggies are undesirable artifacts that we wish to avoid when drawing pixelated lines.

The notion of jaggies applies to drawing pixelated curves as well. Curves are also composed of pixel spans. Figure 2.14c shows an example of a pixelated curve composed of various pixel spans. Each pixel span acts as a small line segment joining the span's diagonal corners, and together the segments form a polygonal path approximation of the

curve. When drawing curves, it is important to preserve curvature. In this case, we are trying to depict a curve with positive curvature and, according to Yu's tutorial,[4] its pixel spans should therefore increase in slope. In Figure 2.14c, the labelled values indicate the slopes of spans and since they form an increasing sequence,[5] the curve satisfies Yu's criterion as a jaggie-free curve.

Figure 2.14d shows an example of a curve that does not form an increasing slope sequence. The sequence increases from $\frac{1}{3}$ to 2, then decreases to 1. The point at which this sequence decreases in slope is called a *jaggie*. It makes the curve look jagged because it violates the curvature requirement. Jaggies are artifacts that pixel artists actively look for and avoid, but that rasterization algorithms ignore. Typical rasterizers will happily produce a curve like the one in Figure 2.14d if it is the closest mathematical approximation of the underlying vector curve. That is why we need special pixelation algorithms to draw paths without jaggies.

To summarize, to draw good line art, we follow three general rules:

1. Paths should be one pixel thick (i.e., they can be represented by slope sequences).
2. Lines should be straight (i.e., the slope sequences should be constant).
3. Curves should have the correct curvature (i.e., the slope sequences should be increasing for curves with positive curvature, and decreasing for curves with negative curvature).

In Chapter 3, we will introduce other types of artifacts and give precise definitions for them.

2.2.2 Antialiasing

When drawing pixel line art, sometimes it is necessary to apply antialiasing to smooth out jagged regions. However, since pixel art images often have low resolutions, it is important to keep the outlines thin. In particular, we want to limit the number of pixels used for antialiasing because using too many will cause the outlines to look thick and blurry. Another

[4] From Yu's tutorial: "For curvature, make sure that the decline or incline is consistent all the way through."

[5] A sequence $\{a_n\}_{n=1}^{\infty}$ is *increasing* if $a_n \leq a_{n+1}$ for all $n \in \mathbb{N}$. It is not necessarily *strictly increasing*, which requires $a_n \leq a_{n+1}$ for all $n \in \mathbb{N}$. Similarly, a decreasing sequence is not necessarily strictly decreasing.

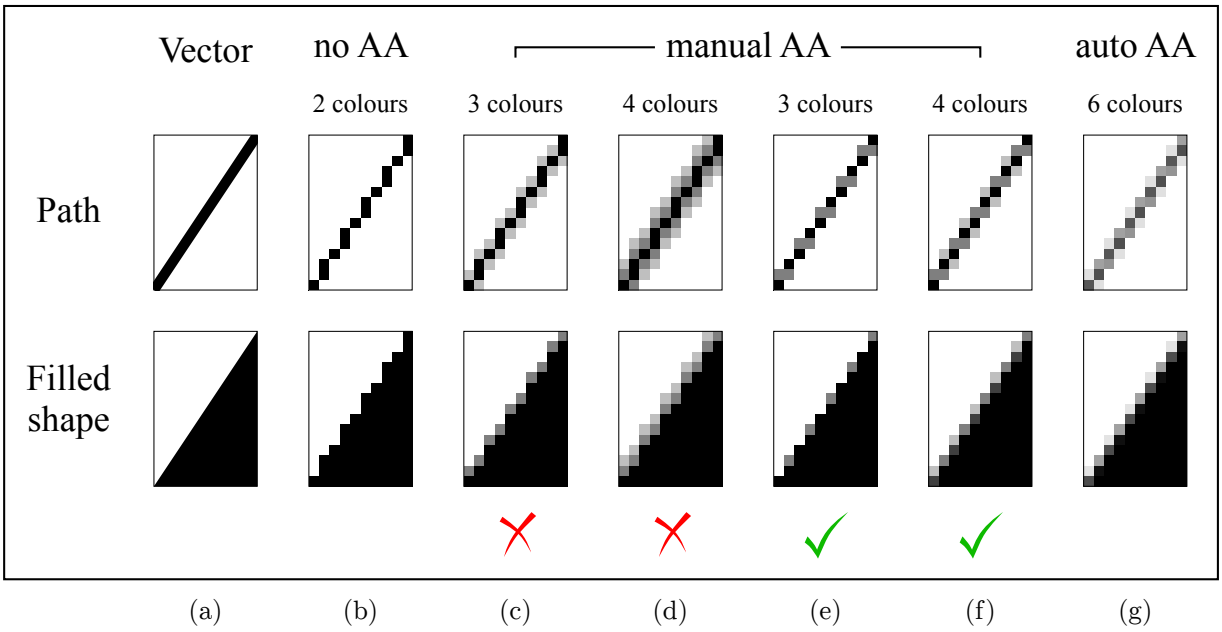


Figure 2.15: A comparison between various ways to draw a one-pixel-thick path and a filled shape. Along with the (a) vector images, we show the difference between (b) no antialiasing, (c–f) various attempts at manual antialiasing, and (g) automatic antialiasing by supersampling. Used with permission.

constraint is the palette size; pixel artists use a limited number of colours for antialiasing so as not to introduce too many new colours to the artwork. This style of antialiasing is called *manual antialiasing* because it is typically done by hand. The examples used in this section are taken from Sven Ruthner's antialiasing tutorial.[6]

Figure 2.15 shows a one-pixel-thick path and a filled triangle rasterized in various ways. Figure 2.15a is the vector input. Figure 2.15b shows rasterization without antialiasing. Since the slope of the diagonal edge is $\frac{3}{2}$, an imperfect slope, the rasterization contains a repeating pattern of jaggies. Figures 2.15c and 2.15d show two bad attempts at manual antialiasing that use too many pixels, creating lines that look too wide and blurry. Figures 2.15e and 2.15f show two successful applications of manual antialiasing that make the lines smoother without introducing too much blur.

Figure 2.15g shows the result of rasterization with *automatic antialiasing*. Automatic antialiasing is a umbrella term used by pixel artists to refer to antialiasing algorithms

[6] http://storage5.static.itmages.ru/i/11/0912/h_1315823707_4054244_c0e6ef162c.png

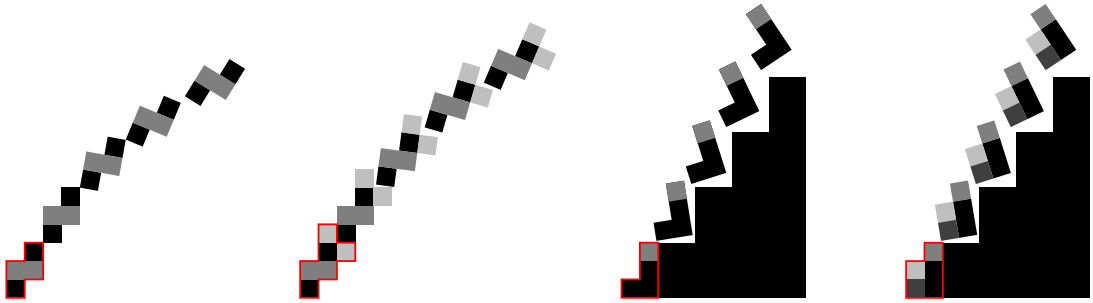


Figure 2.16: Manual antialiasing often contains repetitive pixel patterns. The repeated units are outlined in red.

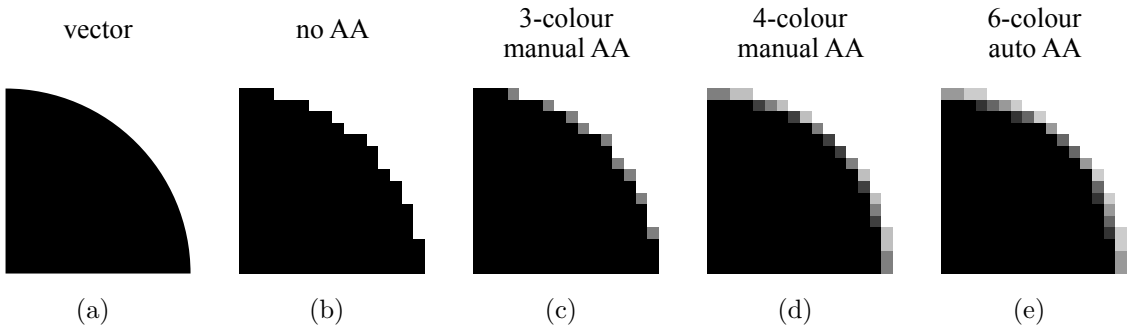


Figure 2.17: A quarter arc of a filled circle: (a) in vector form, (b) rasterized without antialiasing, (c–d) with manual antialiasing, and (e) with automatic antialiasing by supersampling. Used with permission.

that can be applied automatically in image editing tools. In this case, the algorithm used is supersampling. Pixel artists rarely use automatic antialiasing in their art because the rasterized line art is usually too blurry for pixel art and uses too many colours.

Another difference between manual antialiasing and automatic antialiasing is that, when applied to straight lines, manual antialiasing tends to produce more regular pixel patterns. Figure 2.16 shows several manually antialiased lines and diagonal edges of triangles constructed by stacking together many pixel units.

Pixel artists also manually antialias curves, but it is more difficult to do since they do not contain regular pixel patterns. Figure 2.17 shows a quarter arc of a filled circle rasterized without antialiasing, with manual antialiasing, and with automatic antialiasing. Notice that the artist can control how much antialiasing to apply depending on the size

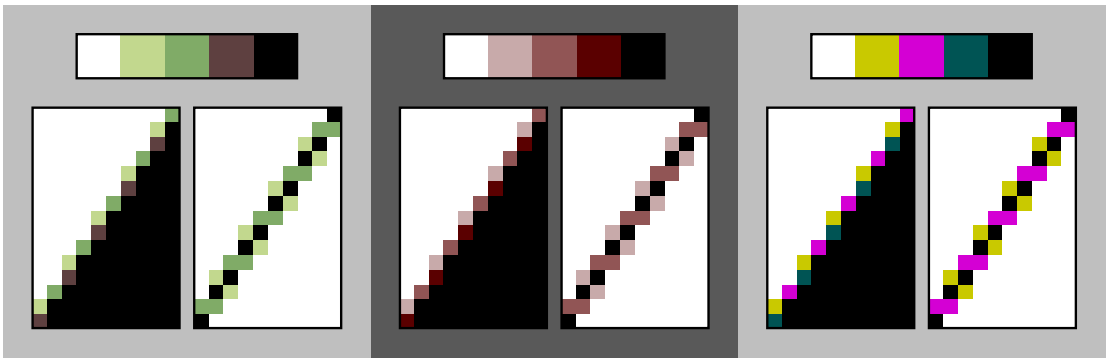


Figure 2.18: Manual antialiasing with non-greyscale colours. Used with permission.

of the colour palette and the desired smoothness of the curve. Even though the two arcs in Figures 2.17d and 2.17e appear to be similar in terms of smoothness, the manually antialiased one uses fewer colours.

Manual antialiasing need not use only greyscale colours. With non-greyscale colours, manual antialiasing is applied based on the brightness of the colours. For example, a bright yellow and a dark brown may be treated as a light grey and a dark grey, respectively. Figure 2.18 shows three different non-greyscale palettes used to antialias a filled triangle and a line. Antialiasing with colours is useful when the artist does not want to introduce new colours to an existing palette.

Even among good examples of manual antialiasing, there are still variations depending on the composition and the artist's preference. These variables include the number of colours used and the amount of antialiasing applied (i.e., the number of pixels used). In Figure 2.19a, for example, only some edges receive antialiasing, while in Figure 2.19b, all the edges are antialiased, with an unusually high number of colours for manual antialiasing. In contrast, the pixel art image in Figure 2.19c uses so few colours that dithering (i.e., alternating pixels of two colours) is used to approximate the effect of antialiasing.

2.3 Related Research in Computer Graphics

Now that we know how artists approach the problem of creating pixel art, we also want to learn about it from a computer scientist's perspective. We will first give an overview of previous work in rasterization and antialiasing, followed by a discussion of font rasterization as it relates to low-resolution pixel art. Finally we will cover research explicitly related to pixel art as a medium.

(a) (b) (c)

Figure 2.19: Variations in manual antialiasing for pixel art: (a) selective antialiasing with low colour count, (b) complete antialiasing with high colour count, and (c) antialiasing by dithering with low colour count. Used with permission.

2.3.1 Rasterization

Rasterization is a fundamental and venerable problem in computer graphics for which many algorithms have been developed. Bresenham introduced simple algorithms for rasterizing lines [10] and circles [11] without antialiasing. His methods have since been extended to handle ellipses [56] and spline curves [4]. Recent research in rasterization is focused more on improving existing algorithms in terms of efficiency either by simplifying the calculations required [9] or by exploiting graphics hardware [42]. Manson and Schaefer [44] recently presented a generalized method for rasterization using wavelets that produces antialiased results in both 2D and 3D.

Antialiasing is a rasterization technique for suppressing aliasing artifacts in image rendering to create a smoother look. There are many antialiasing algorithms, including supersampling, multisampling [3], and fast approximate antialiasing [43]. Recently, Jimenez et al. [37] developed a technique combining morphological antialiasing with multi/supersampling strategies to generate high-quality antialiasing with a fast execution time. Lines and circles can be rasterized more efficiently using Wu's antialiasing algorithm [60, 61].

Although many efficient algorithms exist, not much progress has been made on improving the aesthetic quality of the rasterized output. Despite rasterization algorithms producing many artifacts considered unacceptable for pixel art, no effort has been made

to address this problem. This lack of progress is because most graphics hardware and displays can support such high resolutions and colour counts that pixel-level artifacts are barely visible and considered to be of little consequence. However, pixel art is much more than a by-product of technological constraints; it has evolved into an art form and many games, for example, purposely decrease the resolution by enlarging the pixels to recreate the pixelated look. For this reason, there is still a need for rasterization algorithms to perform well at low resolutions.

2.3.2 Font Rasterization

Although pixel-level details are largely ignored by existing rasterization methods, they are important in the area of font rasterization, where small details can greatly affect the readability of text. In older systems, fonts were stored as bitmaps pre-drawn at various fixed sizes, much like the way icons are designed. This bitmap representation is severely limiting and tedious to design. In most modern systems, fonts are represented by a vector description of the outlines—called *glyph outlines*—and displayed at different sizes via font rasterization, either with or without antialiasing. At sufficiently large font sizes, font rasterization is essentially regular rasterization. However, at small font sizes, maintaining clarity and readability is a challenge and requires special techniques.

Font rasterization involves three main steps: outline grid fitting, outline scan-conversion, and filling [30]. The last two steps are performed using a flag fill algorithm [1] based on the idea that any pixel whose centre lies inside the glyph outlines is considered an interior pixel. The flag fill algorithm flags interior and exterior pixels that correspond to an outline before filling in the pixels in each row.

The first step, outline grid fitting, is based on deforming parts of the outline and adapting them to the grid [29]. *Grid constraints* or *font hints* specify how to perform these outline modifications to preserve properties such as symmetry, stem thickness, and uniformity among similar characters (see Figure 2.20a). In his book, Hersch [30] describes two types of font hints: basic and advanced. Basic hints keep characters aligned via reference lines [8], preserve stem symmetry, and produce discrete arcs of acceptable quality [29]. Advanced hints use snapping and dropout control (see Figures 2.20b and 2.20c) to ensure regularity and character continuity at low resolutions.

The previously discussed techniques in font hinting apply to TrueType [6, 52], an outline font standard developed by Apple that is now ubiquitous in graphical devices. In direct competition is Adobe's Type 1 fonts [32], which use different hinting strategies involving optimal distribution of grid rows and columns within the glyph outline. ClearType uses

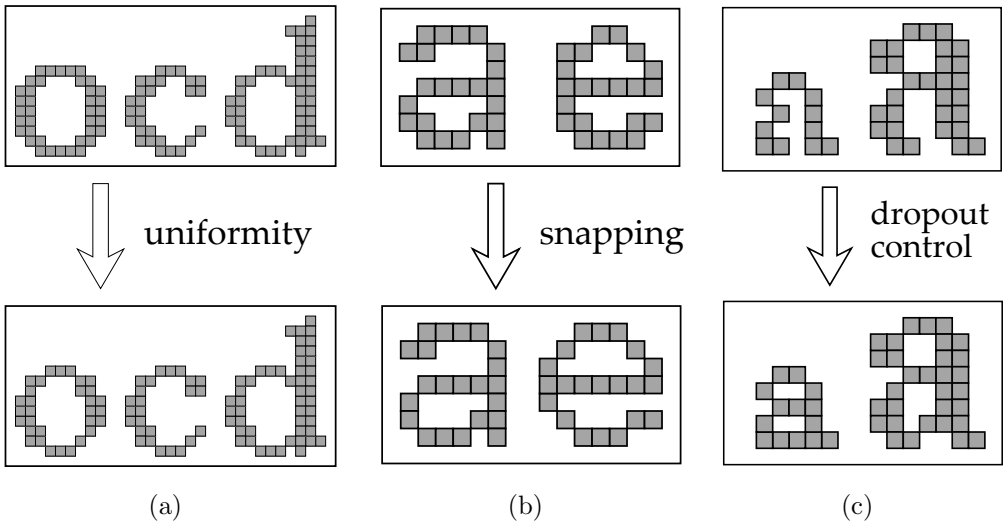


Figure 2.20: Types of font hints: (a) maintaining uniformity across similar glyphs, (b) snapping, and (c) dropout control.

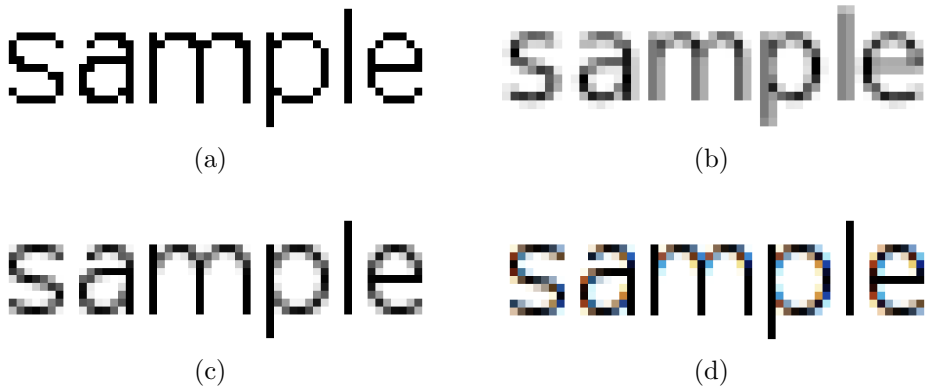


Figure 2.21: Font rasterization (a) without antialiasing, (b) with antialiasing but without hinting, (c) with antialiasing and hinting, and (d) with hinting and subpixel rendering on a RGB flat panel display.

subpixel rendering [7, 17], which takes advantage of the colour subpixel layout in a liquid crystal display to increase the apparent resolution available for antialiasing. Figure 2.21 shows examples of text rasterized with font hinting and subpixel rendering.

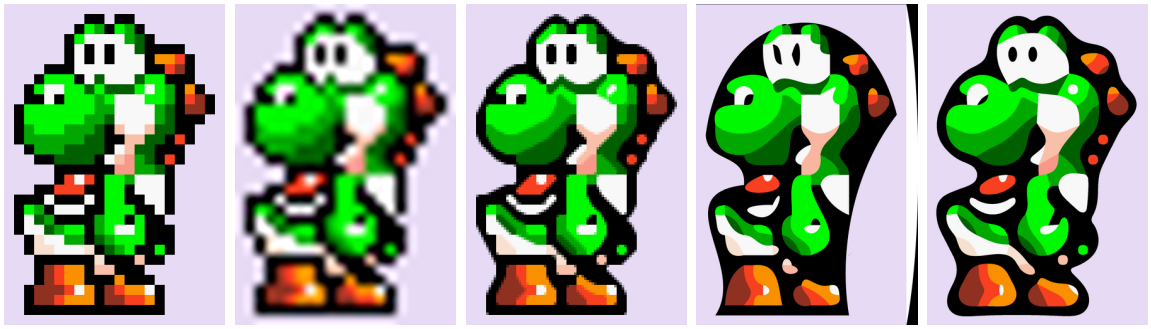
While font hinting significantly improves font rasterization, it is a tedious manual process that can only be done well by an experienced typographer. Some research has been done to improve the efficiency of font hinting. Zongker et al. [63] developed a method for transferring hints from a hinted source font to an unhinted target font. There is also work done in automatic generation of font hints based on models [31] and of gridfitting hints [5]. Knuth's Metafont provides an alternative method of describing vector fonts in terms of geometrical equations [39].

2.3.3 Pixel Art Research

Our goal is to develop rasterization techniques that are better suited to creating pixel art. Traditional rasterization algorithms provide a good starting point and some of the concepts used in font rasterization, such as grid fitting, are applicable to low-resolution rasterization. Currently there are no algorithms that can replace the deft hand of a pixel artist, but with the recent resurgence of retro pixel art games, the computer graphics community has approached pixel art as a research topic from various perspectives.

One of the problems with low-resolution pixel art images is that they cannot be easily resized. Details are easily lost or modified if we apply standard image scaling algorithms to them. Pixel art scaling algorithms is a class of image scaling algorithms that focuses on resizing pixel art images faithfully [58]. Generic image scaling algorithms such as nearest-neighbour (see Figure 2.22a) and bicubic interpolation (see Figure 2.22b) perform poorly on pixel art images because they either augment the aliasing effect or blur the image. Pixel art scaling algorithms create better-looking results for pixel art, but the magnification factors are usually limited to 2× (the most common), 3×, and 4×. The most well-known of these algorithms are Eagle (by Dirk Stevens), 2xSaI [18], Scale2x [45], and the hqx family [53], which handles 2×, 3× and 4× magnification. Figure 2.22c shows the result of hq4x applied to a pixel sprite.

In addition to upsampling algorithms, pixel art scaling algorithms also include vectorization algorithms that convert pixel art into vector art. As expected, generic vectorization algorithms such as Potrace [50], diffusion curves [48], Adobe Live Trace [2] (see Figure 2.22d) and Vector Magic [57] are not as effective when applied to pixel art. Kopf and Lischinski [40] developed a depixelization algorithm specifically for pixel art, which is fast enough to be used in emulators (see Figure 2.22e). The ability to vectorize pixel art images is useful for game studios that wish to re-create an old game's artwork at higher resolutions to re-release it on modern hardware.



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

Figure 2.22: Results from various pixel art scaling algorithms: (a) nearest-neighbour interpolation, (b) bicubic interpolation, (c) hq4x, (d) Adobe Live Trace, and (e) depixelization by Kopf et al.

The opposite problem of creating low-resolution pixel art from high-resolution inputs has also been studied. Gerstner et al. [22] presented an optimization-based downsampling algorithm that converts high-resolution images into pixel art. Their algorithm attempts to optimize both pixel structure and colour palette to retain important features and colours in the original image. Their work has since been extended to allow for more user control [21, 23]. Figure 2.23a compares their results to downsampling by nearest neighbours. More recently, Kopf et al. [41] introduced a content-adaptive image downscaling algorithm that optimizes the shape and locations of the downsampling kernels. In doing so, their algorithm can effectively downscale many types of images, including producing pixel art from vector graphics.

While Gerstner's work is a significant contribution to pixel art research, we are more interested in creating pixel art from a vector input so that components within the image can be transformed and edited with ease, and we do not have to worry about dealing with a noisy raster input image. Our first paper on this subject [35] describes Pixelator, an algorithm that converts vector paths to pixel line art, taking into consideration various artifacts pixel artists try to avoid (see Chapter 3 for more detail). Superpixelator [36], an extension of the previous work, deals with problems that arise when rasterizing shapes such as ellipses that require symmetry preservation (see Chapter 4 for more detail). The new algorithm uses optimization techniques to achieve a good balance between various competing aesthetic goals. In addition, it includes manual antialiasing as a rasterization option (see Chapter 5 for more detail). Figures 2.23b and 2.23c compare Adobe Illustrator's built-in rasterizer to Pixelator and Superpixelator respectively.

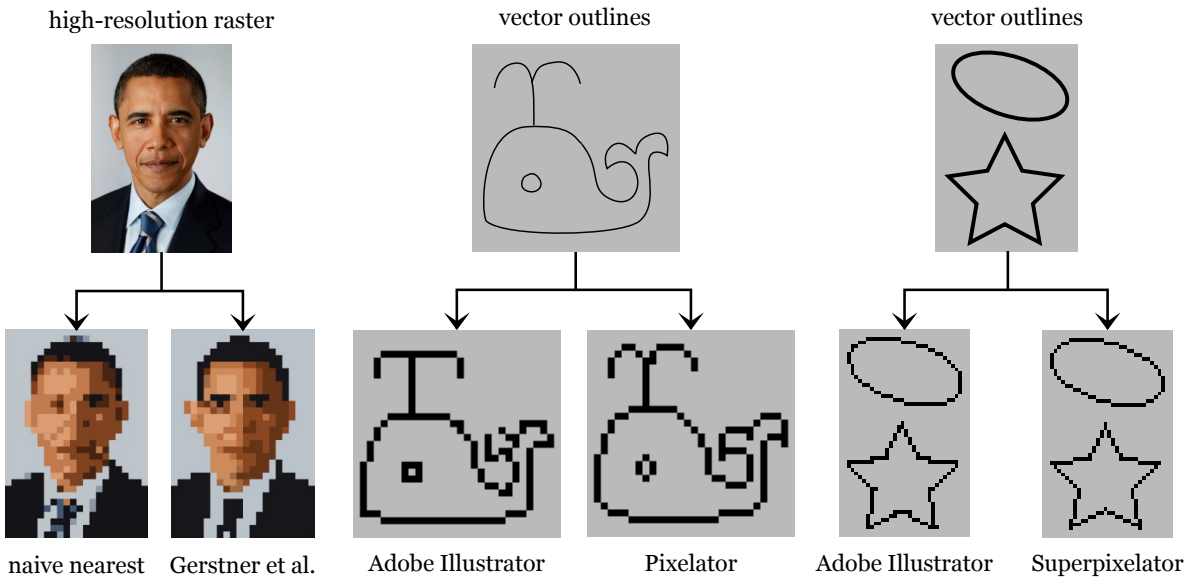


Figure 2.23: Pixel art algorithm comparisons: (a) naïve nearest neighbour downsampling versus the pixelated image abstraction algorithm by Gerstner et al., and (b,c) Adobe Illustrator's rasterizer versus Pixelator and Superpixelator by Inglis et al.

2.4 Summary

Pixel art comes in various styles and a typical workflow consists of many steps, including outlining and antialiasing. Line art is an important component of pixel art that should be drawn as single-pixel-thick and jaggie-free paths. Pixel artists use manual antialiasing to smooth out jaggies without making the lines to look too thick or blurry. Currently, there are no rasterization algorithms that can produce results that meet pixel art standards. However, some ideas in font rasterization can help improve low-resolution rasterization. There has also been work done in other aspects of pixel art, such as upscaling and vectorizing pixel art images, and creating low-resolution pixel art images from high-resolution raster inputs. Our focus is to develop pixelation algorithms that take into account pixel art artifacts when rasterizing vector drawings to create quality pixel art drawings.

Chapter 3

Pixelating Paths

3.1 Challenges in Rasterizing Line Art

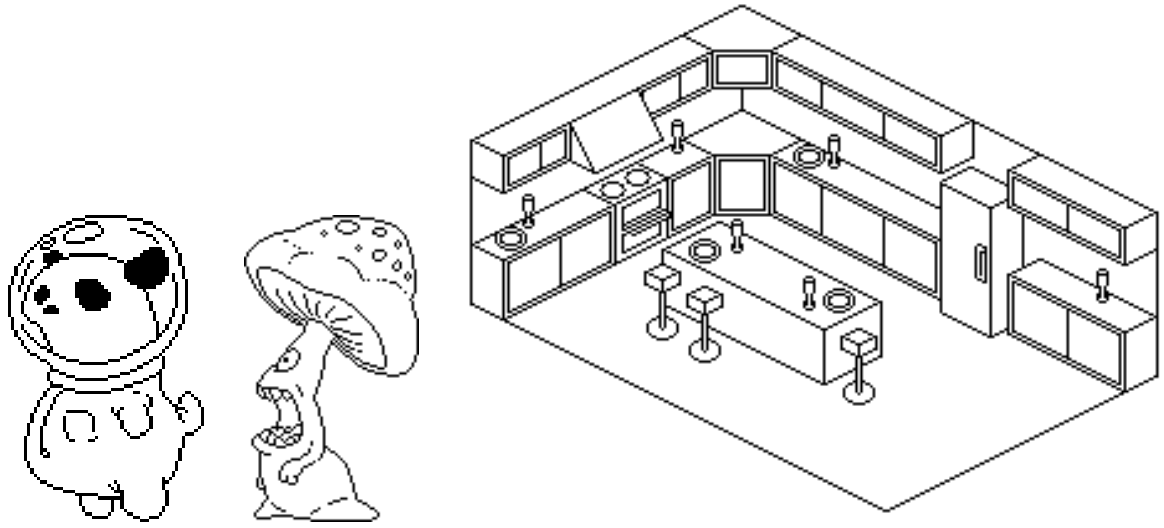
Line art refers to the outlines in an image. In vector graphics, the line art consists of paths described in terms of cubic Bézier splines, and in raster graphics, it is composed of pixels. In pixel art, it is important to draw line art in a "clean" way. Figure 3.1 shows three examples of pixel line drawings created by pixel artists.[1] Notice that the paths are all drawn at one-pixel thickness, lines look straight due to having perfect slopes, and the pixels are arranged in a way that depicts curves as smoothly as possible.

Although line art compositions appear simple, they are created by painstakingly placing each pixel by hand, similar to stippling. Many tutorials discuss how to avoid artifacts when drawing line art. We categorize the artifacts into four types:

1. dropouts;
2. L-shaped corners (and pixel clusters);
3. jaggies; and
4. blips.

For each artifact, we will give a precise definition, discuss when it occurs in regular rasterization, and provide a solution for how to avoid it. Combining these solutions, we produced an algorithm called Pixelator that successfully suppresses most rasterization artifacts.

[1] More pixel line art can be found on Arachne's website: http://retinaleclipse.com/pixelart.html



(a) (b) (c)

Figure 3.1: Pixel line art: (a) *Panda astronaut* by Arachne, (b) *One-eyed mushroom monster* by Arachne, and (c) *Interior of a kitchen* by Rhys Davies. Used with permission.

3.1.1 Dropouts

Dropouts create discontinuities in a pixelated path; they are pixels that should be drawn but are not. Dropouts can occur if, for example, we try to rasterize a Bézier curve by sampling it at parameter values that are too sparse. Of course, this is a naïve method of rasterization, and we expect most rasterization algorithms to be more sophisticated.

Surprisingly, even sophisticated commercial software occasionally produces rasterized paths with dropouts. Figure 3.2 shows two nearly identical ellipses rasterized in Adobe Illustrator CS5.1. Notice that the left ellipse has a dropout at its bottom-left corner while the right ellipse is continuous throughout. The only difference between the two ellipses is how they are positioned relative to the underlying pixel grid, suggesting an inconsistent response to different alignments.

Some rasterization algorithms are guaranteed to not produce dropouts. Bresenham's line algorithm [27], for example, draws a straight line from the starting pixel to the ending pixel by moving at most one pixel horizontally and one pixel vertically at a time. In doing so, the pixels drawn form a continuous line.

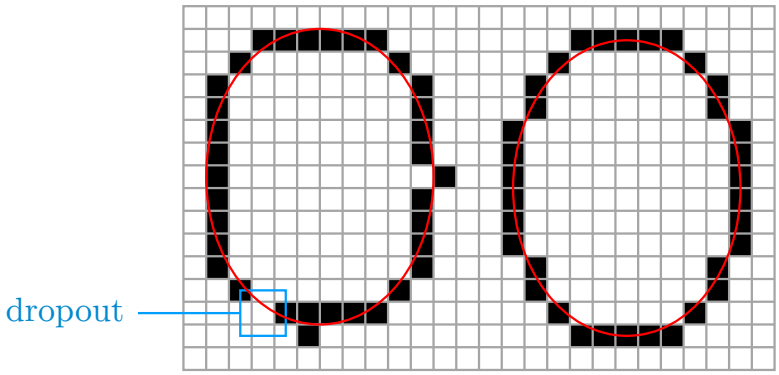


Figure 3.2: The left ellipse has a dropout in its bottom left. The right ellipse does not have dropouts. The red lines indicate the vector paths before rasterization. Images produced with Adobe Illustrator CS5.1.

To avoid dropouts when rasterizing a path, we first approximate it by a polygonal path. Then, for each line segment in the polygonal path, we apply Bresenham's line algorithm. This method is simple to implement and guarantees there are no dropouts in the final rasterization.

3.1.2 L-shaped Corners and Pixel Clusters

Rasterizing a path produces a set of pixels. An *L-shaped corner* is a pixel in this set with at least one horizontal neighbour and one vertical neighbour also in this set. Figure 3.3a shows examples of such configurations. Figures 3.3b and 3.3c show, respectively, a curve with no L-shaped corners and a curve with one L-shaped corner. When depicting a smooth curve, pixel artists avoid L-shaped corners because they make the curve look sharp.

L-shaped corners also appear to stand out. One possible explanation, inspired by Gestalt Laws of Perceptual Grouping [15], is that L-shaped corners look visually different from the other spans that form the curve. As a result, it is difficult to view them all as one continuous object. Figure 3.3d uses different colours to show the perceptual groupings.

L-shaped corners occur if, for example, we try to rasterize a Bézier curve by sampling it at various t-values, but the samples are taken too densely. Bresenham's algorithm does not produce any L-shaped corners, but if we apply it to individual line segments within a polygonal path, then L-shaped corners may occur between adjacent segments. To overcome this problem, we first identify all the L-shaped corners in the pixelated path,

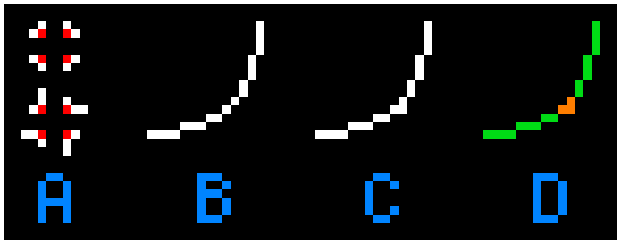


Figure 3.3: (a) L-shaped corners (in red), (b) a curve without L-shaped corners, (c) a curve with one L-shaped corner, and (d) perceptual groupings of such a curve.

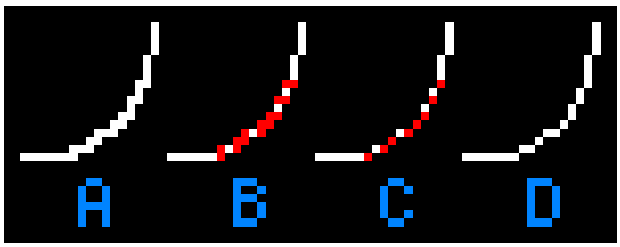


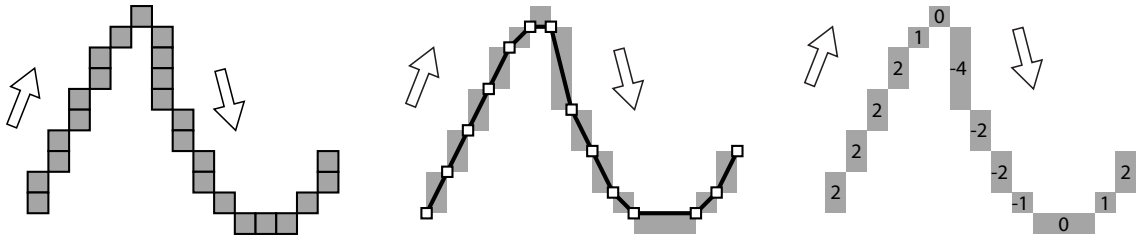
Figure 3.4: (a) A curve with L-shaped corners. (b) The L-shaped corners are identified in red. (c) A subset of the L-shaped corners are removed. (d) The curve after L-shaped corner removal.

and then remove a subset of them, as shown Figure 3.4. We remove only a subset of the L-shaped corners because removing them all at once may render the curve discontinuous. We remove the L-shaped corners that are farthest from the input vector curve first, so that the resulting curve remains a good approximation of the vector curve.

Notice that by removing L-shaped corners from the curve in Figure 3.4a, we get the much thinner-looking curve in Figure 3.4d. We sometimes refer to a group of L-shaped corners as a *pixel cluster* because they form a cluster around a one-pixel-thick path, causing it to look thicker.

3.1.3 Jaggies

In Section 2.2.1, we compared several examples of lines and curves to pixel art to gain a better understanding of how modifying pixel-level features can significantly alter our perception of an image. In particular, we introduced the idea of pixel spans, compared perfect lines to imperfect lines, and discussed what jaggies mean for lines and curves.



(a) (b) (c)

Figure 3.5: (a) A pixelated path with a specified direction, (b) a polygonal path that joins the corners of pixel spans, and (c) the slope sequence that represents the pixelated path.

In this section, we will provide a more rigourous definition of jaggies in terms of slope sequences.

Recall that a pixel span is either horizontal or vertical, and has a length that is equal to the number of pixels it contains. Since a pixelated path is formed by joining a series of pixel spans at their corners, we want to develop a slope sequence notation to describe a pixelated path in terms of the slopes of its pixel spans.

Consider the pixelated path in Figure 3.5a. First we assign it a direction as indicated by the arrow. Then we mark all the corners where adjacent pixel spans are connected, and draw a polygonal path connecting them in order, as shown in Figure 3.5b. Each span has a slope given by the slope of the line segment inside it. The slope values are always of the form n or $\frac{1}{n}$, where n is an integer. Thus the entire pixelated path can be described by the sequence $\{2, 2, 2, 2, 1, 0, -4, -2, -2, -1, 0, 1, 2\}$, as shown in Figure 3.5c.

What we have just described is an ideal case in which the ordering of the pixel spans is clearly defined, but this is not always the true. Figure 3.6a shows a pixelated path whose pixel span order is more difficult to interpret due to the pixel cluster in the circled region. To simplify the problem, we can split the path into two subpaths as shown in Figure 3.6b. Now each subpath can be written as a slope sequence—the left one is $\{2, 2, 2, 2, 2\}$ and the right one is $\{-6, -2, -1, -\frac{1}{2}\}$.

The fact that both subpaths have well-defined pixel span orderings is due to a property called *monotonicity*. Suppose we have a path parametrically defined as $p(t) = (x(t), y(t))$ for $t \in [0, 1]$. We say it is *monotonic in* x if either $x'(t) \geq 0$ for all t, or $x'(t) \leq 0$ for all t (i.e., it is either always increasing in x or always decreasing in x). Monotonicity in y is defined similarly. A path is said to be *monotonic* if it is monotonic in both x and y. Such a path does not contain local extrema except at its endpoints.

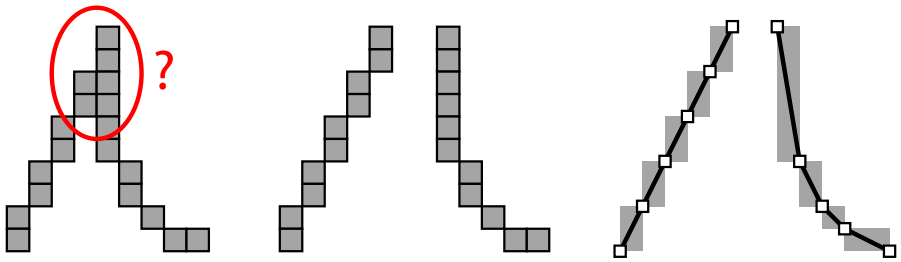


Figure 3.6: (a) A pixelated path with a pixel span ordering that is difficult to interpret due to the pixel cluster (circled). (b) We split it into monotonic subpaths so that (c) the pixel span ordering for each subpath is well-defined.

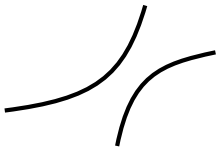
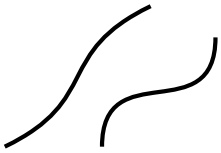
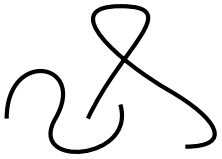
	monotonic	not monotonic
slope-monotonic		
not slope-monotonic		

Figure 3.7: Examples of paths demonstrating monotonicity and slope-monotonicity.

Monotonicity can also refer to the slope. A path is *slope-monotonic in* x if either $x''(t) \geq 0$ for all t, or $x''(t) \leq 0$ for all t (i.e., its slope is either always increasing or always decreasing). Slope-monotonicity in y is defined similarly. We say a path is *slope-monotonic* if it is slope-monotonic in both x and y. Such a path does not contain inflection points.

Figure 3.7 demonstrates the concepts of monotonicity and slope-monotonicity with several examples. For a pixelated path, the definition can be applied to its polygonal approximation. For example, the two subpaths in Figure 3.6b are both monotonic and

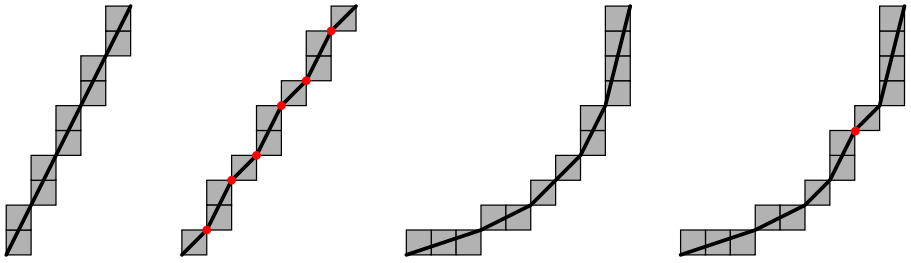


Figure 3.8: Four paths along with their slope sequences to show the difference between a smooth and a jagged path: (a) a straight line with no jaggies, (b) a jagged line with six jaggies, (c) a smooth curve with no jaggies, and (d) a jagged curve with one jaggie. The jaggies are indicated by the red dots.

slope-monotonic. Monotonicity and slope-monotonicity are important because if a pixelated path satisfies both properties, then its pixel span ordering is always unambiguous.

For a vector path that is both monotonic and slope-monotonic, its jaggies are well-defined. We can give a precise definition for jaggies in terms of the slope sequence notation. Suppose we have a vector path that has positive curvature (i.e., $d^2y/dx^2 \geq 0$) everywhere and a pixelated representation of it with the slope sequence $\{m_i : 0 \leq i \leq n\}$. Then the number of jaggies in the pixelated path is defined as

$$\sum_{i=0}^{n-1} \mathbf{1}_{\{m_i > m_{i+1}\}}, \tag{3.1}$$

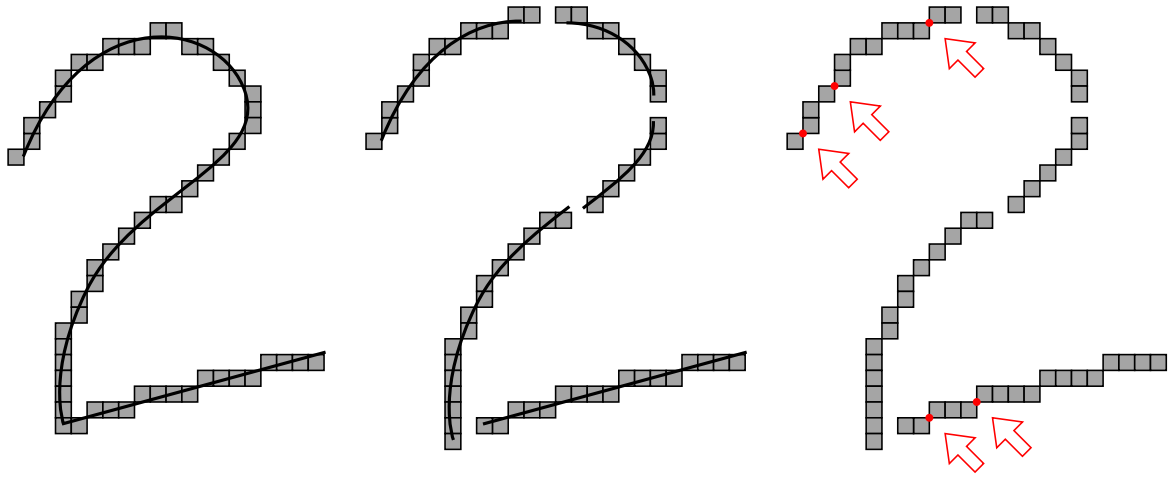
where $\mathbf{1}_A$ is an indicator function that evaluates to 1 if A is true, and 0 otherwise. A jaggie-free path, in this case, satisfies $m_0 \leq m_1 \leq \cdots \leq m_n$.

Jaggies are defined with respect to some curvature. The previous definition is for paths with positive curvatures. For negative curvature paths, the number of jaggies is given by

$$\sum_{i=0}^{n-1} \mathbf{1}_{\{m_i < m_{i+1}\}}. \tag{3.2}$$

For zero curvature paths (i.e., lines), the number of jaggies is given by

$$\sum_{i=0}^{n-1} \mathbf{1}_{\{m_i \neq m_{i+1}\}}. \tag{3.3}$$



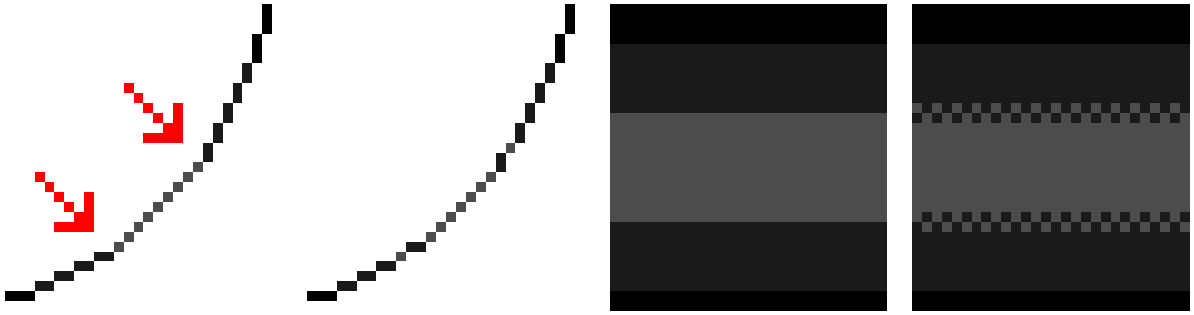
(a) (b) (c)

Figure 3.9: Take (a) a vector path and its corresponding pixelated path, (b) split them into subpaths that are both monotonic and slope-monotonic, and (c) identify the jaggies in each subpath. The jaggies are indicated by the red dots.

Figure 3.8 demonstrates what pixelated lines and curves look like with and without jaggies.

So far, jaggies are only defined for paths with monotonic slope sequences. For a generic path, the number of jaggies it contains is calculated by first splitting the path into monotonic subpaths with monotonic slope, and then taking the sum of the number of jaggies in each subpath. Figure 3.9b shows the splitting process. The jaggies are marked in Figure 3.9c.

Jaggedness, or the presence of jaggies, offers a simple way to measure the smoothness of a pixelated curve. Generally, curves with fewer jaggies look smoother than those with more jaggies. However, there are exceptions to this rule. A jaggie-free curve may not look smooth due to angles created by long pixel segments. For example, in the curve in Figure 3.10a, the transition from a sequence of length-1 spans to a sequence of length-2 spans creates a noticeable angle. However, by adding jaggies between the two segments, we can soften the transition (see Figure 3.10b). Note that this problem only occurs in transitions between $\frac{1}{2}$ and 1 or 1 and 2, and is much less of a problem for any other transition between adjacent slope values. Even when transitioning between $\frac{1}{2}$ and $\frac{1}{3}$, the slope difference is only $\frac{1}{6}$. The method of adding jaggies to soften slope transitions is similar to dithering, which softens the transition between two differently coloured regions (see Figures 3.10c and 3.10d).



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

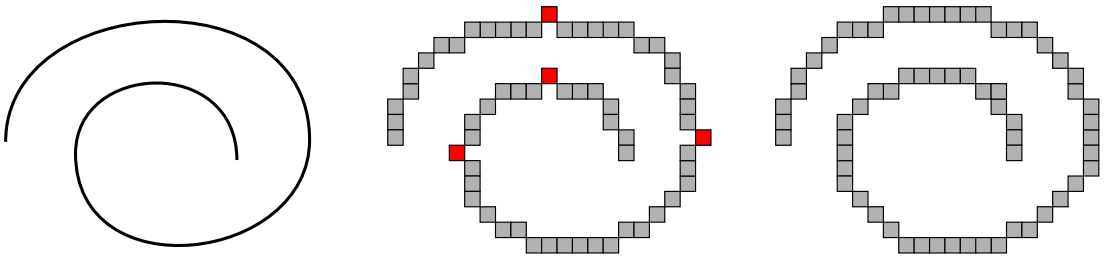
Figure 3.10: (a) The transition from a sequence of length-1 spans to a sequence of length-2 spans can create a noticeable angle (indicated by the arrows). (b) To soften the angle, add jaggies as buffers between the two segments. (c, d) This solution is analogous to using dithering to soften the transition between two differently coloured regions.

Jaggedness is also not a good quality measure for lines because when pixelating an imperfect line, there is no way to remove jaggies unless we significantly change the slope of the line. In Chapter 6, we will explore alternative quality measurements and rasterization algorithms tailored to straight lines.

3.2 Grid Alignment and Blips

A *blip* is an artifact that occurs as a result of bad grid alignment. It is a single pixel that sticks out of a smooth curve or line, causing it to look pointy. For example, Figure 3.11a shows a vector curve rasterized with blips (see Figure 3.11b; the blips are the red pixels) and without blips (see Figure 3.11c).

Blips occur when a vector path is rasterized without any consideration for its grid alignment. Figure 3.12 shows how Adobe Illustrator CS5.1 rasterizes a 12×12 circle with slightly different horizontal offsets. The second row shows a close-up of the rightmost curve segment for each circle, as it moves from straddling two pixel columns to being centred on one pixel column. The third row shows the circles rasterized without antialiasing. Starting with the leftmost rasterization which contains a conspicuous blip, the raster shapes gradually shift towards more horizontally symmetric representation until they reach the rightmost one, which is completely symmetric.



(a) (b) (c)

Figure 3.11: (a) A spiral-shaped vector path, (a) rasterized under bad grid alignment, creating unwanted blips (marked in red), and (b) rasterized under good grid alignment, producing a smooth pixelated curve.

Bad grid alignment can also cause problems in antialiasing. For example, if a vertical vector line straddles two pixel columns, then by supersampling, it will be rasterized as a two-pixel-thick line at 50% opacity, rather than a one-pixel-thick line at 100% opacity, which is obviously the better choice for pixel art. In Figure 3.12, the fourth row shows the effect of grid alignment on antialiasing. Due to misalignment, the leftmost rasterization suffers from blurriness, where the rightmost rasterization has a much more clearly defined boundary.

In both the aliased and antialiased cases, the rasterization looks the best when the four sides (i.e., left, right, top, and bottom) of the circle are completely within a pixel column or row, rather than straddling two pixel columns or rows. In general, given a path to rasterize, we want to align it in such a way that all the local extrema in the horizontal direction avoid column straddling, and all those in the vertical direction avoid row straddling.

Alignment correction cannot be done with translation alone, because simply translating an entire path may remove alignment issues in some places but introduce new ones elsewhere. A better approach is to selectively shift parts of a path, which will require a certain amount of distortion. In Section 3.3.2, we will describe how to align a path to the grid with minimal distortion in a way that removes blips from the resulting rasterization. Later, in Chapter 5, we will introduce a manual antialiasing algorithm that also uses the grid alignment described in this chapter.

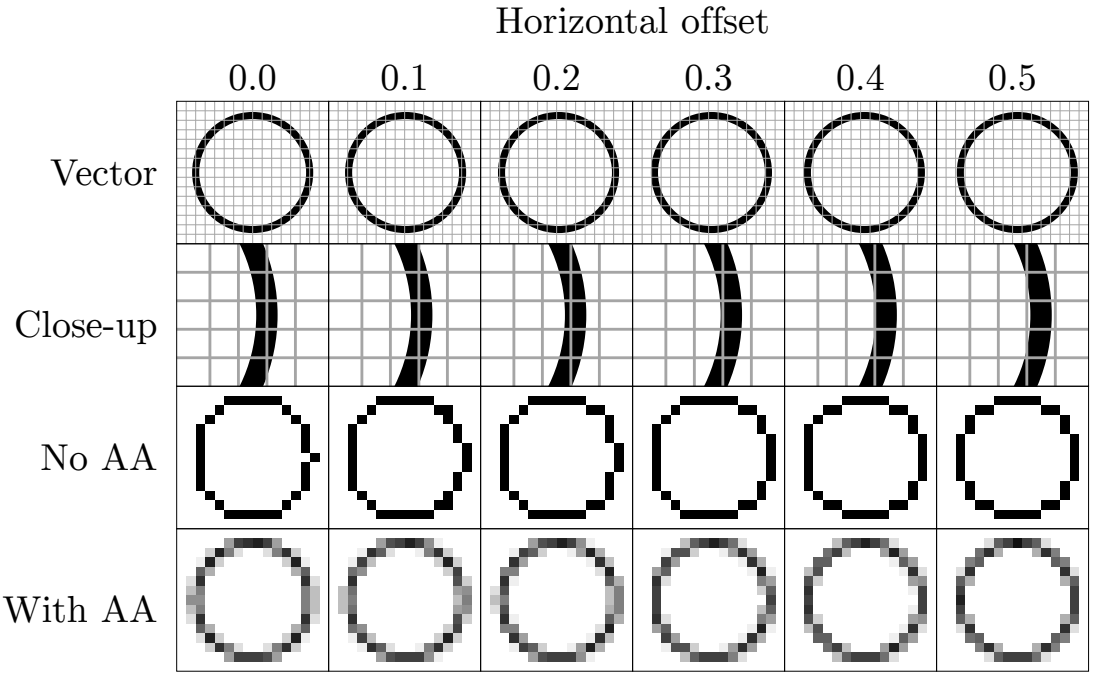


Figure 3.12: Different horizontal offsets are applied to the circles in each column. The table shows how the offset affects the position of the vector circle and its rasterization, both with and without antialiasing. Created with Adobe Illustrator CS5.1.

3.3 Algorithmic Approach

So far, we have defined four different types of artifacts: dropouts, L-shaped corners, jaggies, and blips. In this section, we introduce Pixelator, our pixelation algorithm designed to suppress these artifacts. Since the definition of jaggies requires splitting a path into several subpaths, each of which is both monotonic and slope-monotonic, our algorithm will also take this modular approach. Here is an outline of the Pixelator algorithm, which takes a vector path (expressed as a cubic Bézier spline) and returns a pixelated path:

1. Split the input path into subpaths.
2. Preprocess each subpath.
3. Naïvely rasterize each subpath.
4. Postprocess each pixelated subpath.

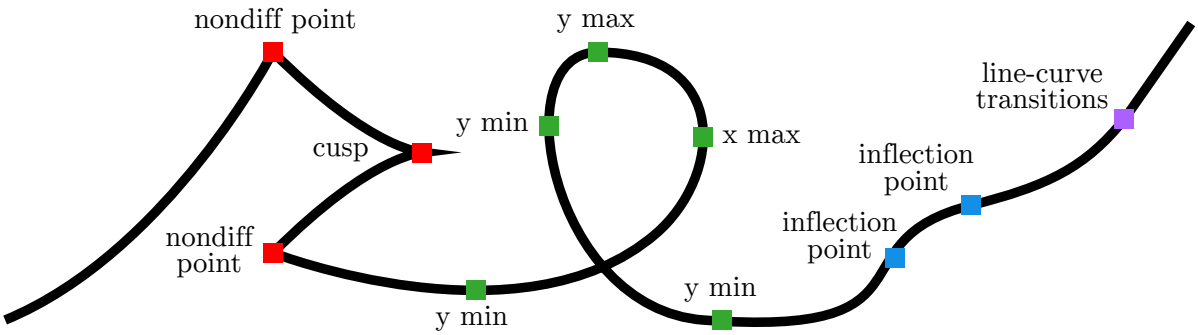


Figure 3.13: A path with non-differentiable points marked in red, local extrema marked in green, inflection points marked in blue, and a line-curve transition point marked in purple.

5. Merge the results to form the final pixelated path.

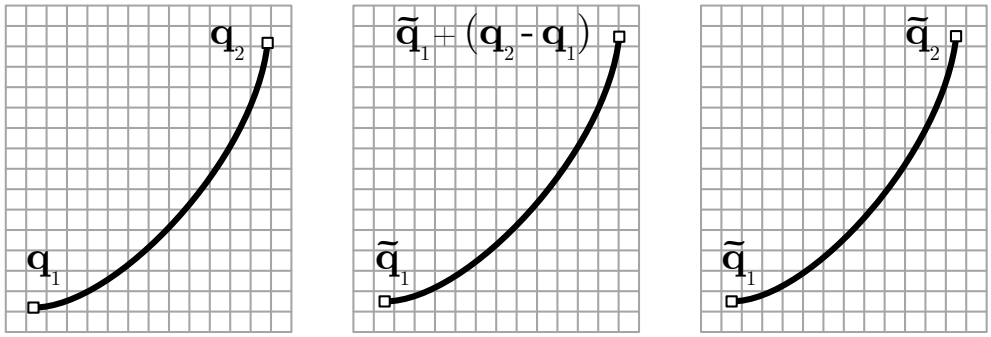
We will describe each step in detail.

3.3.1 Path Splitting

To represent a path using the slope sequence notation, and subsequently identify jaggies in that representation, the path must be split into subpaths that are both monotonic and slope-monotonic. We also want each subpath to be twice-differentiable so that there are no discontinuities in slope. Basically, we want to split the input path at the following points:

1. non-differentiable points (i.e., points that are not twice differentiable),
2. local extrema (i.e., minima and maxima in either the x- or y-direction),
3. inflection points (i.e., points where the sign of the curvature changes),
4. line-curve transition points (i.e., points joining straight lines to curves).

Figure 3.13 shows a path with these points labelled. Methods for calculating these points can be found in standard texts on multivariable calculus [20]. After locating these points, we split the path using de Castlejau's algorithm [19].



(a) (b) (c)

Figure 3.14: (a) Before grid alignment, (b) after $\mathbf{q}_1$ has been shifted to $\tilde{\mathbf{q}}_1$, and (c) after the curve has been properly scaled.

3.3.2 Preprocessing: Shifting for Grid Alignment

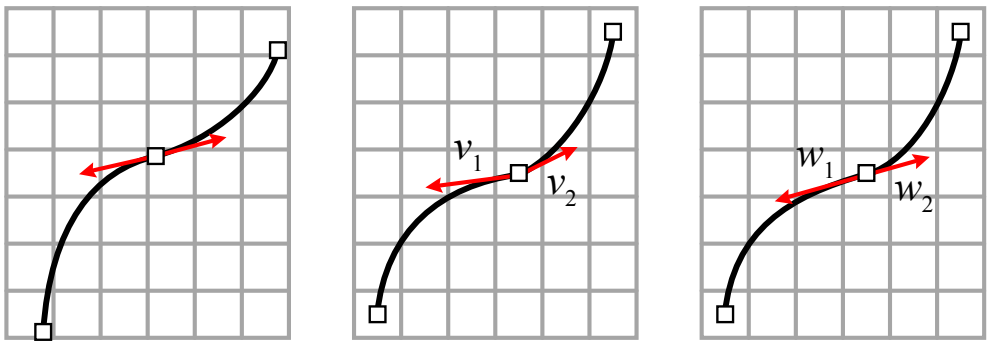
After splitting the input path, the next step is the preprocessing step that involves shifting each subpath so that its endpoints line up on pixel centres. By doing so, we not only eliminate the possibility of creating blips later, but also standardize the path input for the naïve rasterization step described in Section 3.3.3.

To fix the grid alignment for a path, we want to move the endpoints of each subpath to the nearest pixel centre, in a way that creates minimal distortion. Suppose the subpath starts at $\mathbf{q}_1 = (x_1, y_1)$ and ends at $\mathbf{q}_2 = (x_2, y_2)$. Without loss of generality, assume it is increasing. We want to shift endpoints $\mathbf{q}_1$ and $\mathbf{q}_2$ to $\tilde{\mathbf{q}}_1$ and $\tilde{\mathbf{q}}_2$ respectively, where

$$\tilde{\mathbf{q}}_i \quad = \quad (\lfloor x_i \rfloor + 0.5, \lfloor y_i \rfloor + 0.5)\,. \tag{3.4}$$

We perform a translation and non-uniform scaling of the curve to move $\mathbf{q}_1$ and $\mathbf{q}_2$ to $\tilde{\mathbf{q}}_1$ and $\tilde{\mathbf{q}}_2$ respectively. This grid alignment operation has two important properties: (1) the scaling will never stretch the curve by more than one pixel in either x or y, and (2) non-uniform scaling is preferred to rigid motion (i.e., rotation) because it preserves the x- and y-extrema, so that the alignment can effectively remove blips.

When grid alignment is applied to a path, continuity is not affected but some smoothness properties may change. Non-differentiable points are unaffected because they remain non-differentiable. Local extrema with axis-aligned (i.e., horizontal or vertical) tangent vectors are also unaffected because the non-uniform scaling will ensure that the vectors



(a) (b) (c)

Figure 3.15: (a) Before inflection point shifting, and (b) after inflection point shifting, (c) with its tangent vectors readjusted to remove the slope discontinuity.

remain axis-aligned after grid alignment. The only case in which smoothness is affected is when an inflection point is involved. Figure 3.15a shows an example of a path divided at an inflection point into two subpaths. After the endpoints of each subpath are aligned as shown in Figure 3.15b, the tangent slopes on either side of the inflection point are scaled differently, resulting in a slope discontinuity.

To fix the slope discontinuity, we need to adjust the tangent slopes slightly without changing the curves too much. Let v_1 and v_2 be the tangent vector after grid alignment. We replace them with new tangent vectors $w_1 = \frac{||v_1||}{||v_2 - v_1||}(v_1 - v_2)$ and $w_2 = \frac{||v_2||}{||v_2 - v_1||}(v_2 - v_1)$. These two vectors point in opposite directions, and have the same lengths as v_1 and v_2 (see Figure 3.15c). With the new tangent vectors, C^1 continuity is preserved at inflection points.

3.3.3 Naïve Rasterization

We now have a set of subpaths, each of which is smooth, monotonic, slope-monotonic, and has endpoints on pixel centres. The next step is to compute, for each subpath, a simple rasterization that can be written in terms of the slope sequence notation. Being able to express the pixelated subpath in this notation guarantees continuity (i.e., no dropouts) and single-pixel thickness (i.e., no L-shaped corners or pixel clusters). The simple rasterization will most likely produce jagged paths—a problem that we will address in Section 3.3.4

Bresenham's Line Algorithm

Our naïve rasterization is a modification of Bresenham's algorithm. Bresenham's line algorithm is based on the idea that the equation of a line can be used to define an error function

$$e(x, y) \quad = \quad y - (mx + b), \tag{3.5}$$

where (x, y) is on the line if $e(x, y) = 0$, above the line if $e(x, y) > 0$, and below the line if $e(x, y) < 0$. Suppose the line has a positive slope m where $0 \leq m \leq 1$ and we are currently at pixel $\mathbf{p} = (x, y)$. To get the next pixel, we first increment the x-component, and then decide if the y-component should also be incremented. In other words, the two possible candidates for the next pixel are $\mathbf{p}_x = (x + 1, y)$ and $\mathbf{p}_{xy} = (x + 1, y + 1)$ with respective error values of $e_x = e(x+1, y)$ and $e_{xy} = e(x+1, y+1)$. If $|e_{xy}| < |e_x|$, then the y-component is incremented.

To compute the error, notice that as x is incremented, the error changes by $dx = e(x + 1, y) - e(x, y) = -m$ and as y is incremented, the error changes by $dy = e(x, y + 1) - e(x, y) = 1$. This method of tracking error by calculating the error difference is called *forward differencing*. Figure 3.16 shows an example of Bresenham's algorithm used to draw a line segment of slope $m = \frac{4}{5}$. Starting from the left endpoint whose error is zero, we consider two candidates with error values of $e_x = -m = -\frac{4}{5}$ and $e_{xy} = 1 - m = \frac{1}{5}$, and choose the one with the least absolute error. Repeat until the right endpoint is reached.

If the line has a positive slope value greater than 1, then at each step of the algorithm, we always increment the y-component and optionally increment the x-component. In practice, the algorithm can be made more efficient by converting rational coefficients to integers so that only integer calculations are required.

Bresenham's algorithm has been extended to handle cubic Bézier curves (an implementation is provided by Zingl [62]), but it is complicated due to considerations of cusps, inflection points, etc. Even though our subpaths are composed of cubic Bézier curves, they satisfy nice properties such as differentiability, monotonicity and slope monotonicity, which means there are no cusps, inflection points, or local extrema except at the endpoints. Therefore we propose a simpler way to obtain a naïve rasterization: first subdivide a subpath using de Castlejau's algorithm to pixel resolution (i.e., each subdivided segment is no more than one pixel width in length), and then apply Bresenham's line algorithm to draw each line segment in the subdivision. Note that even though this method produces a fairly crude rasterization of the subpaths, the next step in the algorithm will suppress most of the rasterization artifacts.

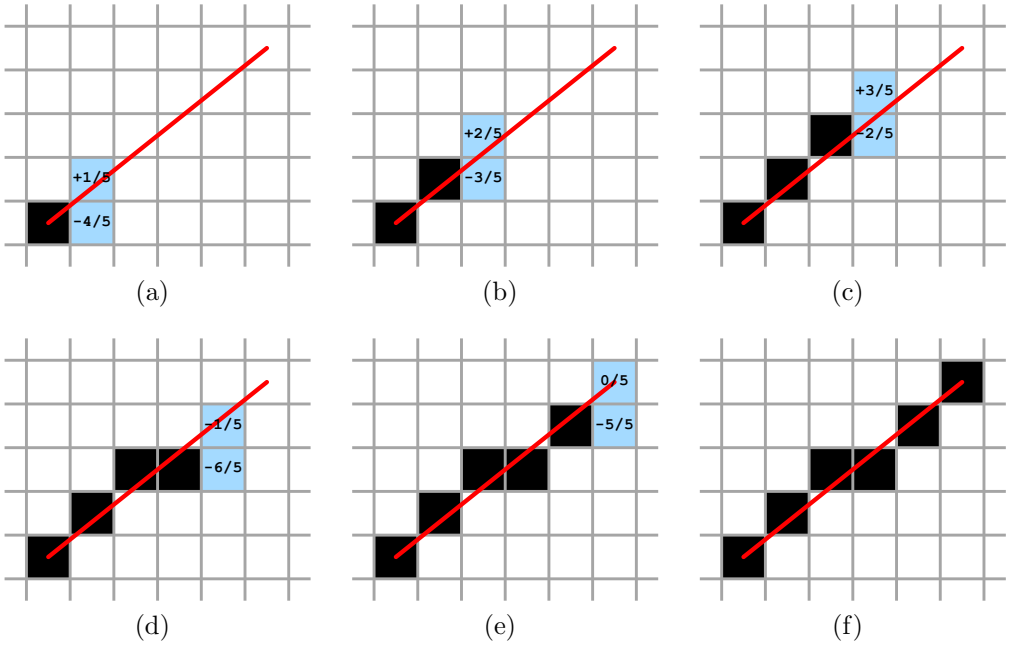
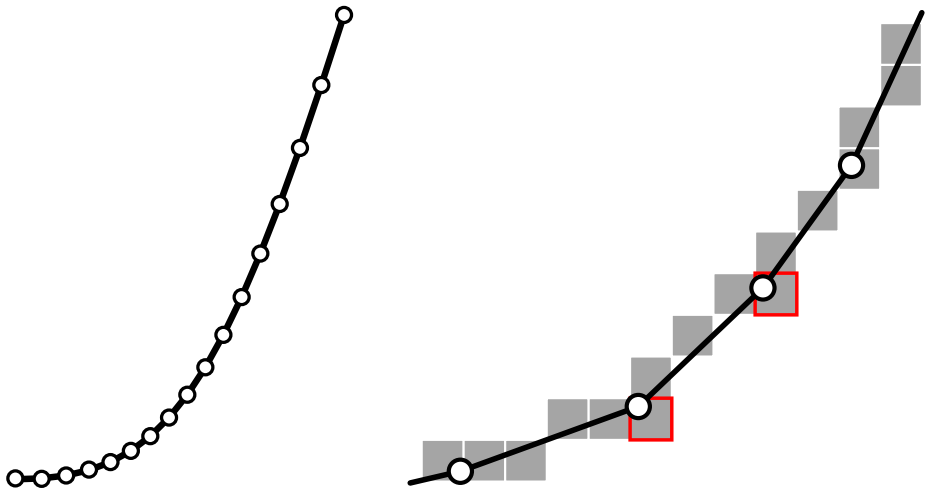


Figure 3.16: Steps in Bresenham's line algorithm.

An example of such a subdivision is shown in Figure 3.17a. During this process, it is possible to encounter L-shaped corners between neighbouring segments, as shown in Figure 3.17b. To remove them without breaking the continuity of the path, we detect all L-shaped corners and remove them in decreasing order of their distances from the path until no more can be removed without creating a discontinuity. We then express the resulting pixelated subpath in slope sequence notation.

3.3.4 Postprocessing: Partial Sorting for Jaggie Removal

The only artifacts left that we should address are the jaggies. Recall that a jaggie-free path is one that can be represented by a monotonic slope sequence (i.e., either increasing or decreasing). Therefore, one simple way to remove jaggies is to sort the slope sequence either in increasing or decreasing order, depending on whether the path has positive or negative curvature.



(a) (b)

Figure 3.17: (a) Apply de Castlejau's algorithm to subdivide a subpath, and then (b) remove L-shaped corners.

The following are a few examples of this slope-sorting method. Figure 3.18a shows a pixelated path given by

$$\{\tfrac{1}{5}, 1, \tfrac{1}{2}, 1, 1, 2, 1, 2, 3, 5\}, \tag{3.6}$$

which contains two jaggies. After sorting it in increasing order, the new path

$$\{\tfrac{1}{5}, \tfrac{1}{2}, 1, 1, 1, 1, 2, 2, 3, 5\} \tag{3.7}$$

is jaggie-free and therefore looks smoother. Figure 3.18b shows another pixelated path given by

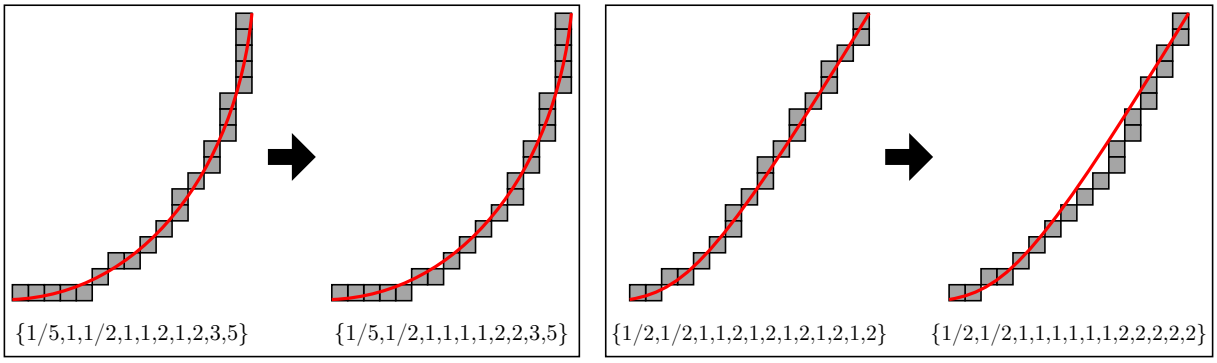
$$\{\tfrac{1}{2}, \tfrac{1}{2}, 1, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2\}. \tag{3.8}$$

After sorting, it becomes

$$\{\tfrac{1}{2}, \tfrac{1}{2}, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2\} \tag{3.9}$$

. Unlike in the previous example, the sorted path in this case is a poor approximation because it deviates from the vector path by almost a pixel in the middle.

The main difference between the two examples in Figure 3.18 is the number of jaggies in the initial unsorted pixelated path. The more jaggies we start with, the more deviation



(a) (b)

Figure 3.18: (a) Sorting the pixelated path $\{\frac{1}{5}, 1, \frac{1}{2}, 1, 1, 2, 1, 2, 3, 5\}$ removes two jaggies to give a smoother look. (b) However, sorting the pixelated path $\{\frac{1}{2}, \frac{1}{2}, 1, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2\}$ creates too much deviation from the vector path.

will be introduced as a result of sorting. Ideally, we want to remove just enough jaggies to improve the smoothness of the pixelated path without creating significant deviation from the vector path.

We propose a *partial sorting* algorithm that partially sorts a slope sequence, taking into account various factors that affect rasterization quality. The idea is to define a cost for each pixelated path in terms of jaggies and deviation such that a lower cost corresponds to a better pixelation. Then we iterate over the pixels in the path, consider a set of minor adjustments, and choose the optimal adjustment. We then repeat this greedy process until the pixel configuration converges.

The cost of a pixelated path is defined in terms of its positional deviation D_p, slope deviation D_s, and sortedness S. Positional deviation refers to the how much the pixelated path deviates from the vector path. Slope deviation is how much the slope of the pixelated path deviates from that of the vector path. Sortedness measures how much of the corresponding slope sequence is in sorted order. Appendix A provides exact definitions and pseudocode for computing each quantity. The overall cost C of a pixelated path is given as a weighted sum $C = 3D_p + 3D_s + S$, in which the weights are determined empirically with the intention of maximizing visual appeal. Figure 3.19 compares pixelation results from using four different cost functions as a demonstration of the effect of the three quantities.

We wish to find the pixelated path that minimizes the cost. However, finding the global minimizer would be too slow since we want our algorithm to run at interactive speed.

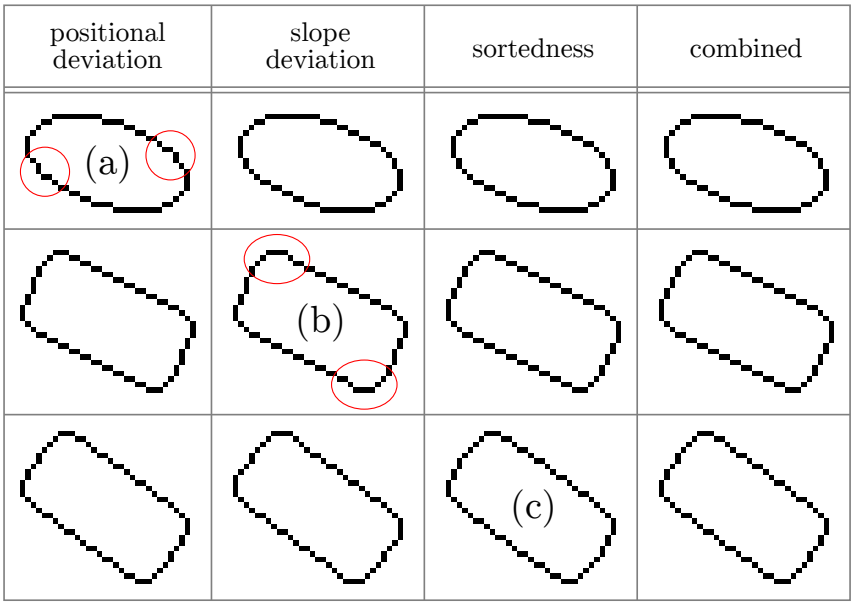
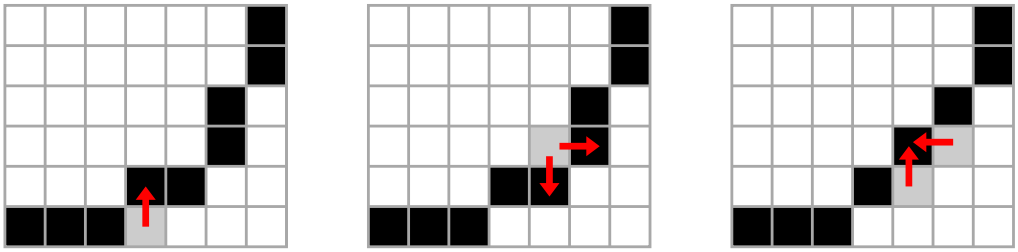


Figure 3.19: A comparison of pixelation results using four different cost functions: positional deviation only ($C = D_p$), slope deviation only ($C = D_s$), sortedness only ($C = S$), and combined ($C = 3D_p + 3D_s + S$). (a) Using only positional deviation may result in jaggies. (b) Using only slope deviation may create overly rounded corners. (c) Using only sortedness may produce slightly bulging shapes. (d) The combined cost function does not produce the aforementioned artifacts.

Instead, we use a greedy approach: given the naïvely pixelated path from the previous step, we check within its neighbourhood and accept the neighbour with the lowest cost if the cost is lower than that of the current pixelated path. We then repeat this process until convergence (i.e., when all the neighbours have higher costs than the current pixelation).

A neighbour of a pixelated path is defined as another pixelated path that can be obtained by applying one of three moves: shifting a pixel (see Figure 3.20a), splitting a pixel into two (see Figure 3.20b), or merging two pixels into one (see Figure 3.20c), as long as the resulting path remains connected.

At each step of our partial sorting algorithm, we need to compute the costs of all the neighbours of a particular pixelated path to compare them. However, since neighbouring paths have many pixels in common, we only need to calculate the difference involving the modified pixels, which greatly reduces the computation time. As for how long it takes for



(a) (b) (c)

Figure 3.20: To find a neighbour of a pixelated path, either (a) shift a pixel, (b) split a pixel into two, or (c) merge two pixels into one.

the algorithm to converge, we tested several monotonic and slope-monotonic paths smaller than 50×50 and, in all cases, the algorithm converged to a local minimum within 10 steps.

3.4 User Study

To test the quality of the results produced by our pixelation algorithm, we designed a user study. The goal of the study was to collect hand-drawn samples of pixel art and compare them to pixel art generated through automated rasterization. The user study consisted of two phases: the Drawing Phase and the Rating Phase. In the Drawing Phase,[2] participants were asked to create pixel art images corresponding to some reference vector drawings. In the Rating Phase,[3] they were given pairs of pixel art images to compare in terms of visual appeal and similarity to the reference image.

3.4.1 Drawing Phase

For this user study, it was important to gather drawing data from participants with pixel art experience. Since there are many active online pixel art communities, we created web applications (using Google Web Toolkit) that allowed users to draw and rate pixel art. For the Drawing Phase, the web application consisted of a questionnaire and a simple drawing tool. The questionnaire collected information about age, gender, artistic level (types of art

[2] http://cgl-pixel-draw-1.appspot.com
[3] http://cgl-pixel-rate-1.appspot.com

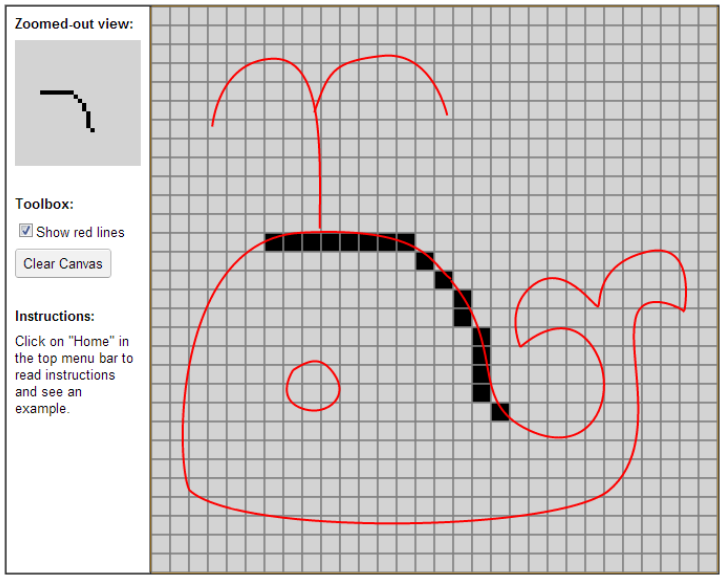


Figure 3.21: A screenshot of the drawing tool used for the Drawing Phase of the user study.

they have practiced in the past), familiarity with pixel art, experience with creating pixel art (how many years of practice), e-mail address, and any other relevant information such as links to samples of their artworks. Appendix C contains the actual questionnaire.

The drawing tool was an HTML canvas with zoomed-in and zoomed-out views (as shown in Figure 3.21). The zoomed-in view showed a red vector line drawing superimposed on a pixel grid. Clicking on a pixel toggled it on or off. We deliberately avoided introducing higher-level tools for drawing lines or curves, because we wanted participants to make a deliberate decision about the fate of each pixel in their drawings. There were eight images in total, shown in Figure 3.22 along with names which we will use to refer to individual images. Four of the images were placed on 20×20 pixel grids, and the other four used 30×30 grids. These images were carefully designed to incorporate a variety of features such as sharp angles, densely packed curves, and facial expressions. We deliberately included some challenging features such as curves that straddle two columns of pixels and lines with imperfect slopes to challenge both the participants and the rasterizers.

The objective was to use the pixels to depict the red vector line images. Participants were specifically told that "There are no right or wrong answers, so just draw what you think looks best! You do not need to follow the red lines exactly." We recorded both the pixels drawn and the order in which they were drawn or erased, to provide some insight

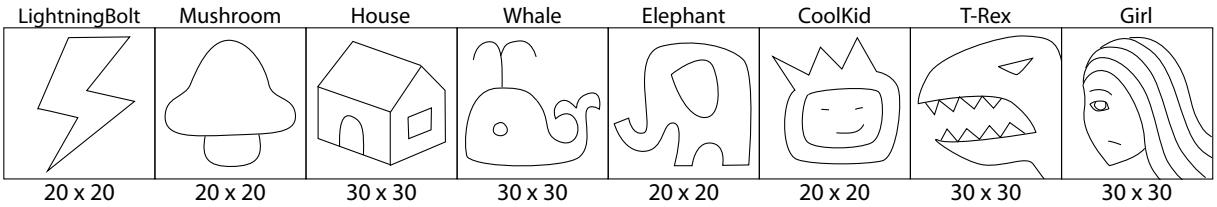


Figure 3.22: Participants were asked to reproduce each of these eight images on a small pixel grid of the given dimensions.

into workflow. To eliminate learning effects, we randomized the order in which the eight images were shown.

3.4.2 Rating Phase

To set up the Rating Phase, we first collected all the hand-drawn pixel art created by participants in the Drawing Phase. A participant's submission was considered incomplete if it provided no demographic information or if one or more of the canvases were completely empty. After discarding the incomplete data sets, we divided the remaining data into groups for stratified sampling.

From the questionnaire data collected in the Drawing Phase, two factors were potentially most relevant to the quality of an individual's pixel art drawing: their artistic level and pixel art experience. Referring to the questionnaire in Appendix C, artistic level is defined as the number of checked items in Question 3, and pixel experience ranges from 0 (for no experience with creating pixel art) to 6 (more than ten years of experience). For each variable, we calculated the median, and then used the medians to split drawings into four groups:

(A) low-art-low-exp (low artistic level and low pixel art experience)

(B) low-art-high-exp (low artistic level and high pixel art experience)

(C) high-art-low-exp (high artistic level and low pixel art experience)

(D) high-art-high-exp (high artistic level and high pixel art experience)

To these four groups we added three more:

(E) Pixelator (our pixelation algorithm)

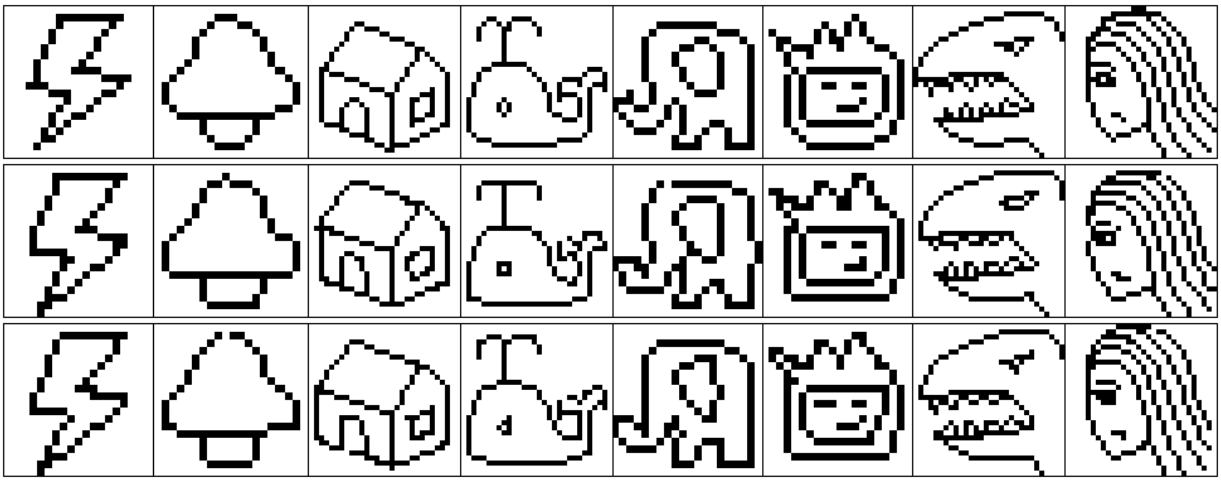


Figure 3.23: The top, middle, and bottom rows contain respectively the images generated by Pixelator, Illustrator CS5.1, and Photoshop CS5.1.

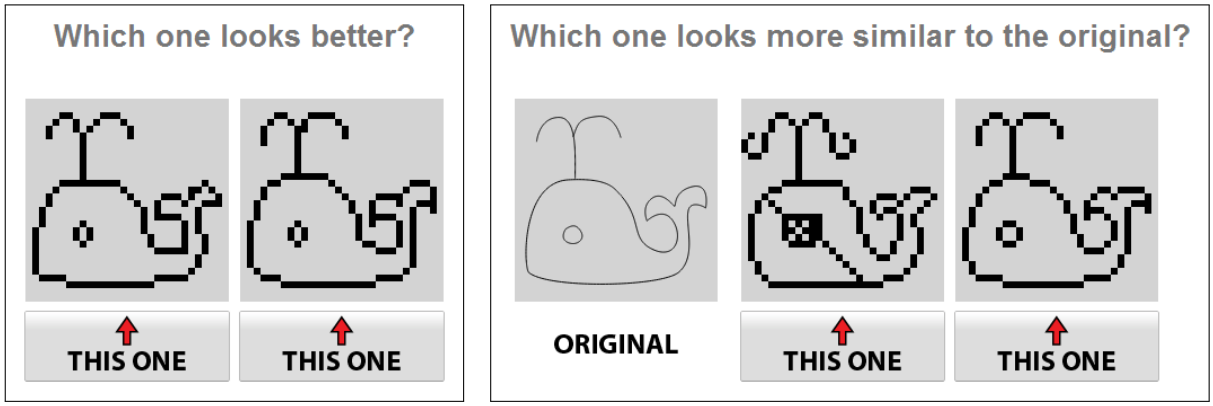
(F) Adobe Illustrator CS5.1 (its built-in rasterization algorithm)

(G) Adobe Photoshop CS5.1 (its built-in rasterization algorithm)

The main difference between the human groups (A–D) and the computer groups (E–G) is that, for each vector image, a computer group has exactly one corresponding pixel image, whereas a human group has multiple images created by various participants that fall in that group. Figure 3.23 shows all the computer-generated pixel art drawings. As for the hand-drawn pixel art images, they were randomly sampled and paired with other images for comparison in the Rating Phase.

In the Rating Phase, participants were asked to compare pairs of pixel art images in terms of visual appeal and similarity to the vector reference images. The image pairs were chosen as follows.

1. Randomly choose one of the eight reference images in Figure 3.22.
2. Randomly choose two different groups from A to G.
3. If Group A, B, C or D is chosen, randomly choose a participant in that group and use the pixel art drawing created by that participant based on the chosen reference image.



(a) (b)

Figure 3.24: Screenshots from the Rating Phase of the user study asking participants to evaluate (a) visual appeal and (b) similarity.

4. If Group E, F or G is chosen, use the pixel art drawing generated by that algorithm based on the chosen reference image.

5. Pair the two images from the two chosen groups.

The Rating Phase consisted of 200 image pairs split into two sets of 100. The first set contained pairs of pixel art images and participants were asked to rate them by visual appeal (see Figure 3.24a). The second set showed the reference image along with each pair of pixel art images and participants were asked to rate them by their similarity to the reference image (see Figure 3.24b). Instead of presenting the image pairs interspersed, participants were presented with one complete set followed by the other set to avoid confusion. To remove bias, the order in which the sets were presented was randomized.

We recruited participants from two sources: our university and online communities. For on-campus recruitment, we invited participants to do the study in our lab, and afterwards asked them questions regarding their experience to collect some qualitative data on the drawing and rating process. For example, we asked some participants why they made certain choices in their pixel art drawings and whether they found some of the image comparisons particularly difficult. For the online recruitment, our goal was to get more pixel artists to participate so that we could compare our result to high-quality pixel art. In addition to advertising on social media websites such as Google+ and Facebook, we posted a link to the pixel art subreddit,[4] a special interest group for pixel art on a popular

[4] http://www.reddit.com/r/pixelart

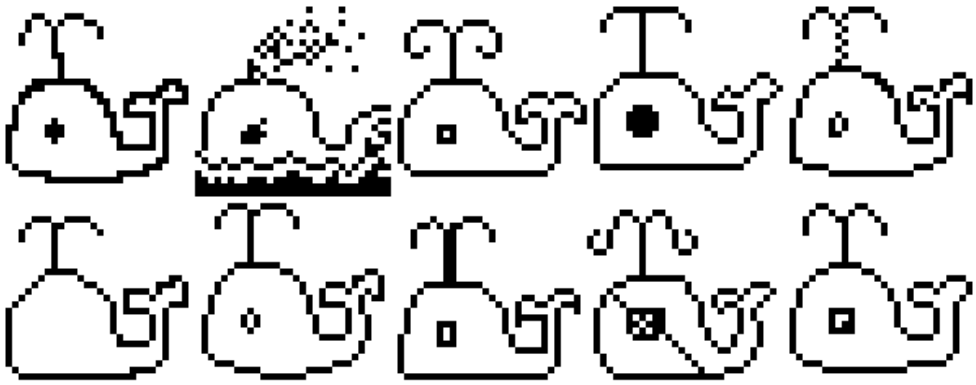


Figure 3.25: Examples of different pixel drawings of the Whale.

link-sharing website, and The Spriters Resource,[5] a community of artists who create pixel art sprites. Since we cannot directly observe the online participants, we tracked their pixel toggling to create playbacks.

3.4.3 Results and Analysis

For the Drawing Phase, there was a total of 148 participants, the majority of whom were male and between the ages of 18 and 27. 129 participants reported to have encountered pixel art in the past, while 90 had actually created pixel art. Out of the 90 who had created pixel art, 55 reported to have more than one year of experience. The complete demographic information of the participants is summarized in Appendix D as bar plots.

For each reference image, there was a large variation among the pixel art images created. To demonstrate the diversity, Figure 3.25 shows ten pixel drawings based on the Whale reference image. They vary from using thinner lines to thicker lines, outlines to filled regions, faithful renditions to creative interpretations, missing features to added features, etc. For each feature of the Whale, people came up with different ways to draw it. For the outline, some people chose a flatter bottom while others drew it more rounded. The eye, the tail and the spout were drawn in various shapes, styles and sizes. Some people added slight modifications to avoid artifacts and others simply traced the vector outline given.

Despite the variation, most of the participants stayed on task and produced reasonable pixel art representations of the reference vector images. Figure 3.26 shows the averages of

[5] http://www.spriters-resource.com

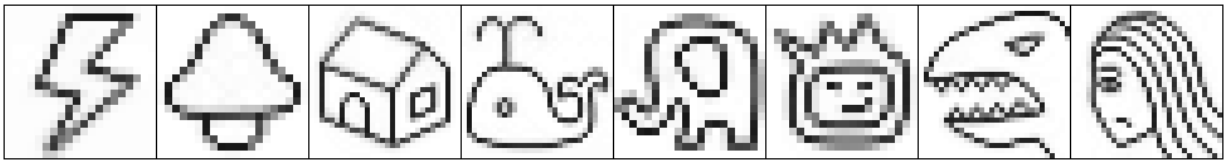


Figure 3.26: The averages of the images collected in the Drawing Phase.

these pixel drawings, calculated by taking the mean of the pixel values. As expected, the averages resemble antialiased rasterizations of the vector drawings. Darker pixels represent unanimous decisions, such as the top edge of the LightningBolt and the vertical edges in the House. Grey pixel clusters, on the other hand, indicate areas of ambiguity, such as the bottom edge of the CoolKid and the top edge of the Elephant. These two examples both contain vector paths that straddle two rows of pixels, thus creating ambiguity.

From observing participants and watching playbacks of their drawing process, we noticed that participants with little experience tended to take the task more literally by tracing the vector outlines with pixels. They usually learned quickly from looking at the zoomed-out view that a faithful rendition does not always produce the best-looking results, and resorted to shifting the paths slightly. Pixel art drawings created by amateurs often contained many artifacts, whereas those with experience typically followed pixel art rules and created one-pixel-thick paths with minimal jaggies and blips. Some participants—usually those with extensive pixel art experience—reinterpreted the reference images to create pixel art images that may be visually appealing but not at all similar to the vector input. Participants that we interviewed agreed that acute angles, lines with imperfect slopes, and densely packed areas were the most difficult features to reproduce, while long curves with low curvature were the easiest.

The Rating Phase had 200 participants overall, some of whom participated in the Drawing Phase as well. For each group, we calculated the proportion of participants that favoured it as being more visually appealing or more similar to the reference image, and summarized the results in Figures 3.27 and 3.28. The values in the bar plots represent the percentage preferred in all image pairs that included an image from the respective group. For example, in Figures 3.27, Group E (Pixelator) has 63%, which means out of all the pairwise image comparisons in which one of the two images is taken from Group E, 63% of time the images from Group E were chosen as more visually appealing.

These percentages serve as rough estimates of how the groups rank in terms of visual appeal and similarity. However, the results are not definitive because there is variation among the frequencies in which each group was chosen as well as how often the two groups

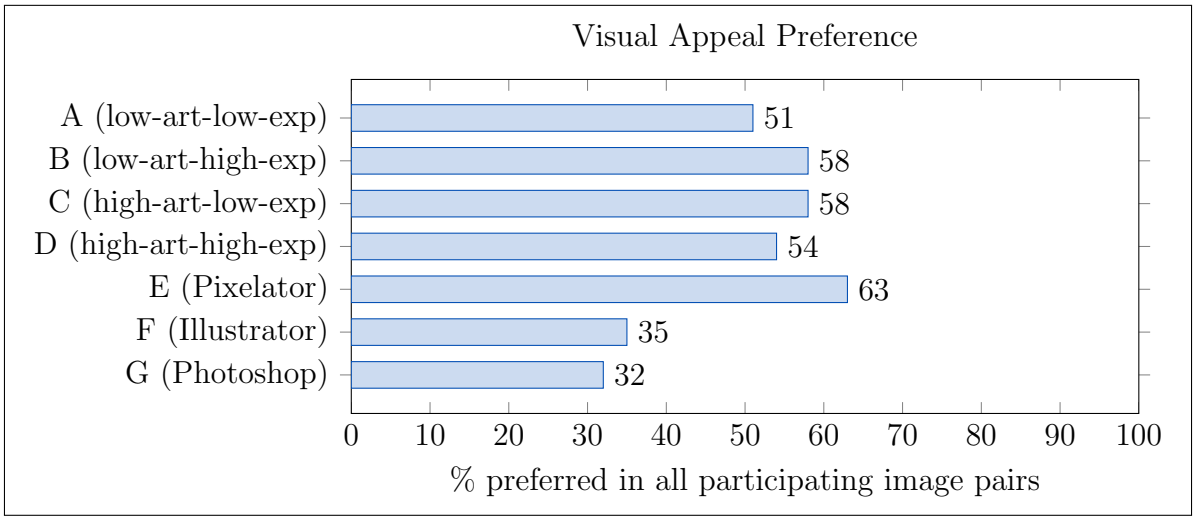


Figure 3.27: A histogram of the visual appeal preference from the Rating Phase of the user study.

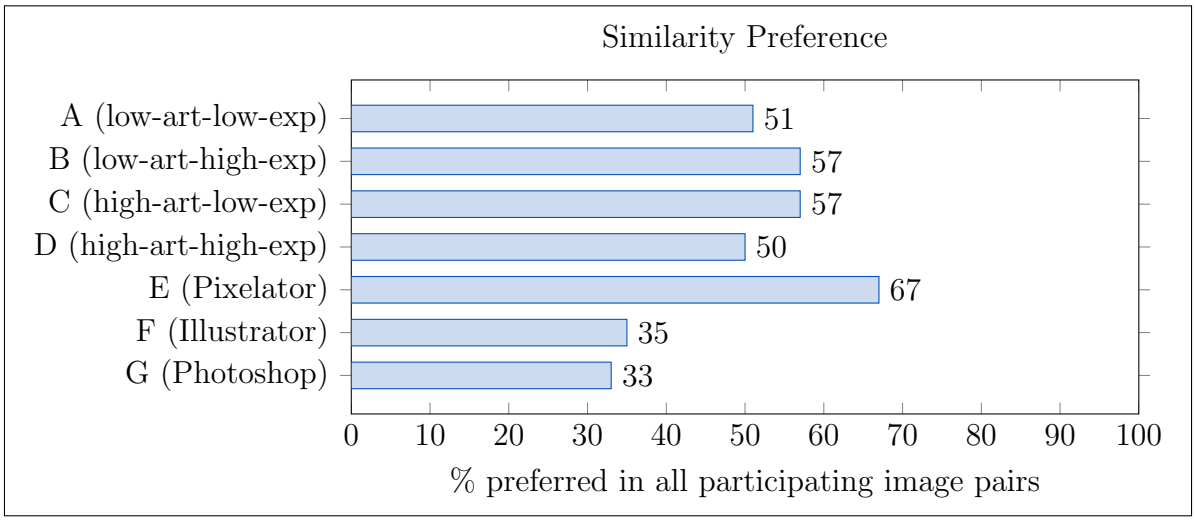


Figure 3.28: A histogram of the similarity preference from the Rating Phase of the user study.

were selected for direct comparison. To determine whether there is statistical significance between the performance of the groups, we conducted a formal comparison between each pair of groups via hypothesis testing. To demonstrate how the comparison is done, suppose

we want to compare Group E (Pixelator) and Group F (Illustrator) in terms of visual appeal based on the collected data. First we extract all the pairwise comparisons that involve these two groups and consider the set of participants who were asked to do at least one of these comparisons. Let $\{x_{i1}, x_{i2}, \dots x_{im_i}\}$ be the set of ratings done by participant i, where $x_{ij} = 1$ means Group E is favoured while $x_{ij} = 0$ means Group F is favoured. Let p_i be the probability that participant i prefers an image from Group E over one from Group F. Let $\bar{p}$ be the average of the p_is, which corresponds to probability that Group E is favoured over Group F by all the participants in terms of visual appeal. We are interested in whether or not there is a statistical significance in the preference. That is, we wish to test the null hypothesis $H_0 : \bar{p} \leq 0.5$ against the alternative hypothesis $H_1 : \bar{p} > 0.5$.

Under the hypothesis, we can deduce using the Central Limit Theorem that

$$\frac{\left(\frac{1}{n}\sum_{i=1}^{n}\hat{p}_i\right) - 0.5}{\frac{1}{n}\sqrt{\sum_{i=1}^{n}\frac{\hat{p}_i(1-\hat{p}_i)}{m_i}}} \xrightarrow{d} \mathcal{N}(0,1)\,. \tag{3.10}$$

where $\hat{p}_i$ is the mean of $\{x_{i1}, x_{i2}, \dots, x_{im_i}\}$. See Appendix B for the formal derivation. Thus, we calculate the corresponding p-value. If it is falls below 0.05, we reject the null hypothesis in favour of the alternative hypothesis that participants prefer the images in Group E over those in Group F based on visual appeal. Tables 3.1 and 3.2 show the p-values from all the pairwise comparisons for visual appeal and similarity respectively. For example, in Table 3.1, if the p-value in row E and column F is less than 0.05, it means the images in Group E are rated as significantly more visually appealing than those in Group F. The p-values less than 0.05 are highlighted to indicate statistical significance. In both tables, the p-values in row E are all highlighted, indicating a significant preference of Group E over all other groups in terms of both visual appeal and similarity.

By conducting these pairwise comparison tests, we can determine between every group pairing if there exists a significant preference of one group over another among the participants. The results may not always produce a simple ordering of the groups based on preference. Using the $>$ symbol to denote "is preferred over", we could for example have a situation in which $A > C$ but neither $A > B$ nor $B > C$, in which case we cannot say $A > B > C$. Luckily, our results happened to produce simple orderings for both visual appeal and similarity (with occasional ties), as summarized in Table 3.3.

These outcomes suggest that our algorithm Pixelator outperforms both Illustrator and Photoshop. The ratings also show that, on average, people prefer images generated by Pixelator to those created by each group of human subjects. However, it is important to

Group	A	B	C	D	E	F	G
A (low-art-low-exp)		0.996	1.000	1.000	1.000	0.000	0.000
B (low-art-high-exp)	0.004		0.692	0.033	0.987	0.000	0.000
C (high-art-low-exp)	0.000	0.308		0.001	0.979	0.000	0.000
D (high-art-high-exp)	0.000	0.967	0.999		1.000	0.000	0.000
E (Pixelator)	0.000	0.013	0.021	0.000		0.000	0.000
F (Illustrator)	1.000	1.000	1.000	1.000	1.000		0.016
G (Photoshop)	1.000	1.000	1.000	1.000	1.000	0.984	

Table 3.1: p-values of pairwise group comparisons for visual appeal. The p-values below 0.05 are highlighted, indicating statistical significance.

Group	A	B	C	D	E	F	G
A (low-art-low-exp)		1.000	1.000	0.450	1.000	0.000	0.000
B (low-art-high-exp)	0.000		0.271	0.002	1.000	0.000	0.000
C (high-art-low-exp)	0.000	0.729		0.001	1.000	0.000	0.000
D (high-art-high-exp)	0.550	0.998	0.999		1.000	0.000	0.000
E (Pixelator)	0.000	0.000	0.000	0.000		0.000	0.000
F (Illustrator)	1.000	1.000	1.000	1.000	1.000		0.101
G (Photoshop)	1.000	1.000	1.000	1.000	1.000	0.899	

Table 3.2: p-values of pairwise group comparisons for similarity. The p-values below 0.05 are highlighted, indicating statistical significance.

Visual Appeal Ranking	Similarity Ranking
1: Group E	1: Group E
2: Group B	2: Group B
2: Group C	2: Group C
3: Group D	3: Group D
4: Group A	3: Group A
5: Group F	4: Group F
5: Group G	4: Group G

Table 3.3: Rankings of visual appeal and similarity based on results from the Rating Phase. When two consecutive groups have the same ranking, it indicates that no statistically significant difference was found in their ratings.

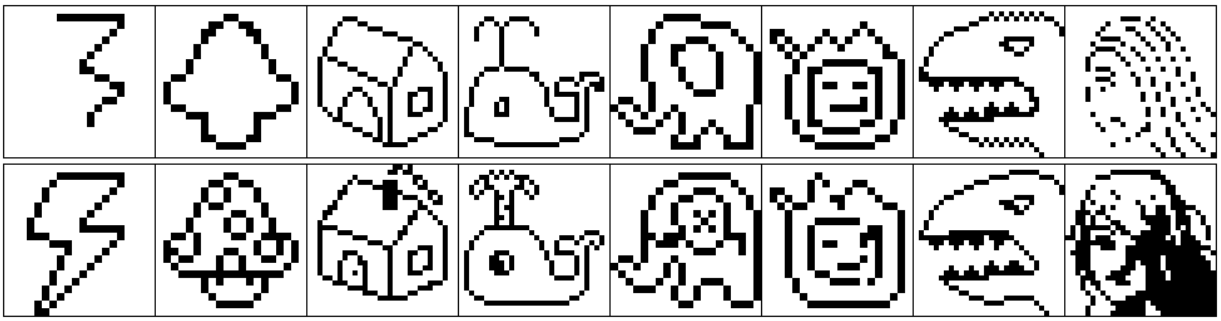


Figure 3.29: The top and bottom rows contain, respectively, hand-drawn images with the lowest visual appeal and similarity ratings.

note that this does not mean our program performs better than all individual pixel artists, but rather that the variables used to split the human subjects are poor predictors of how highly their works are judged. There are a number of participants whose drawings are rated more highly than those generated by Pixelator.

Interestingly, even though we expect participants with high artistic level and more pixel art experience to create images with higher ratings, often this is not the case. In fact, many experienced pixel artists created images that deviated significantly from the reference image, and therefore received low ratings for similarity. We are surprised to find that images that are sufficiently different from the reference image, despite their high quality and creativity, often received low ratings for visual appeal as well (see Figure 3.29). One explanation for this discrepancy is that many participants from the Rating Phase also took part in the Drawing Phase, and knew what the reference images should look like even though they were not shown as part of the comparisons for visual appeal. Participants we interviewed also reported that the more creative images usually look messier, contributing to their lower visual appeal.

Overall, the results of the user study suggest that Pixelator outperforms both Illustrator and Photoshop at pixel art resolutions. Our algorithm also creates better pixel art, on average, than all four human groups. However, there is still room for improvement since there are many hand-drawn images that individually scored higher than those generated by our algorithm. To learn how to further improve our algorithm, we took the data from the Drawing Phase, and calculated the proportion of votes each image received for both visual appeal and similarity. Using these values, we extracted for each of the eight references the most highly rated pixel art drawings in terms of visual appeal and similarity, as shown in Figure 3.30. All of these are hand-drawn images.

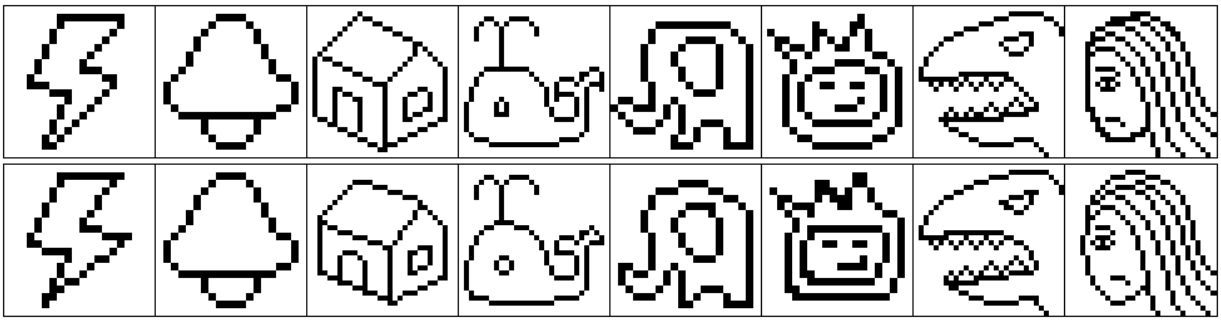


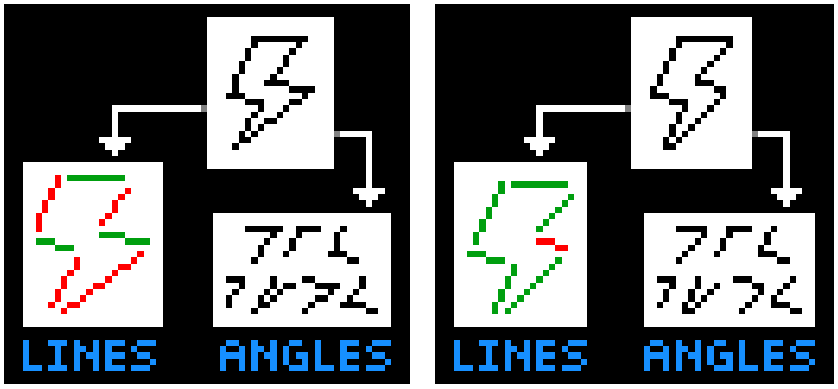
Figure 3.30: The top and bottom rows contain, respectively, hand-drawn images with the highest visual appeal and similarity ratings.

Comparing these highly rated images to those generated by Pixelator, some common features include the use of smooth non-jagged curves and unbroken lines. At such low resolutions, any jaggies and blips are painfully obvious and can be spotted even by amateurs. Looking at the top row of images which are rated highly for visual appeal, we find that certain features have been altered, such as the slope of the straight lines in the LightningBolt and the House, the roundness of the two faces, the size of the Girl's pupil, and the shapes of the T-Rex's teeth. Most of these alterations require processing semantic information about the reference images, which our algorithm does not perform.

Looking at the bottom row of images which are rated highly for similarity, we see that features such as the acute angles in the LightningBolt, and roundness of the Whale's eye, and the curved contour on the Elephant's body have been preserved—even somewhat exaggerated. It is interesting to note the same Mushroom drawing has been rated highest in both visual appeal and similarity. It contains no jaggies, and while there is symmetry in the stem, the mushroom cap is depicted with a subtle asymmetry that accurately reflects the reference image.

3.5 Summary

To summarize, Pixelator is a pixelation algorithm that specializes in low-resolution rasterization of line art. The algorithm contains steps that mimic how pixel artists remove artifacts including dropouts, L-shaped corners, jaggies, and blips. The results of the user study suggest that Pixelator creates more visually appealing rasterizations that more closely resemble vector inputs than images created by Illustrator and Photoshop, or by many experienced pixel artists.



(a) (b)

Figure 3.31: (a) The LightningBolt rasterized by Pixelator contains more imperfect lines (in red) than perfect lines (in green), and the angles appear to be more messy. (b) In contrast, the most highly rated hand-drawn LightningBolt contains almost all perfect lines, and the angles appear more neatly drawn.

One of the things the user study did not test is the ability to draw geometric primitives such as ellipses and rectangles. Currently, these shapes can be parametrized as cubic Bézier splines to use as inputs for our pixelation algorithm. However, Pixelator does not properly rasterize shapes by preserving properties such as symmetry and aspect ratio. In the next chapter, we will describe exactly how Pixelator fails to address these issues and how to improve our algorithm so that shape properties are preserved through rasterization.

Pixelator also lacks the ability to draw visually appealing lines and angles at low resolutions. In Figure 3.31, we took the LightningBolt image rasterized by Pixelator and the hand-drawn one with the highest visual appeal rating, and extracted the lines and angles. By comparing them, we see that the hand-drawn image contains many more perfect lines and neatly drawn angles. In Chapter 6, we will introduce two line-drawing algorithms for drawing aliased and antialiased lines and, in Chapter 7, we will describe an angle-drawing algorithm.

Chapter 4

Pixelating Shapes

Pixelator, as described in Chapter 3, draws paths well, but it does not take into account properties of shapes. Symmetry, for example, is an important property found in many geometric primitives such as ellipses, rectangles, and stars. Given a symmetric vector shape as input, we would like our pixelation algorithm to produce an equally symmetric pixelated output. In addition to symmetry, there are also other shape properties to take into account. In this chapter, we describe an improvement on Pixelator, called Superpixelator, that deals with the preservation of various shape properties. The evaluation involves a qualitative comparison of Superpixelator to other rasterizers and to a professional pixel artist in terms of their ability to rasterize a set of geometric primitives.

4.1 Ellipse Test

We are interested in how various rasterizers handle the preservation of symmetry and other properties. Our test involves rasterizing a set of ellipses subject to multiple transformations including translation, rotation, and scaling. We tested five rasterizers: Adobe Illustrator CS5.1,[1] CorelDRAW Graphics Suite X6, Java 2D, Pixelator, and Superpixelator. We will begin by showing the results from the first four rasterizers, identify their shortcomings, and then show Superpixelator's results for comparison after describing the algorithm.

Figure 4.1 shows the set of ellipses used for the test. All ellipses have 180° rotational symmetry. Set 4 contains axis-aligned ellipses, which have reflection symmetry about their

[1]Photoshop was not tested because it rasterizes geometric primitives the same way Illustrator does. However, this does not contradict the use of Photoshop in Chapter 3's user study because the two programs rasterize generic paths differently.

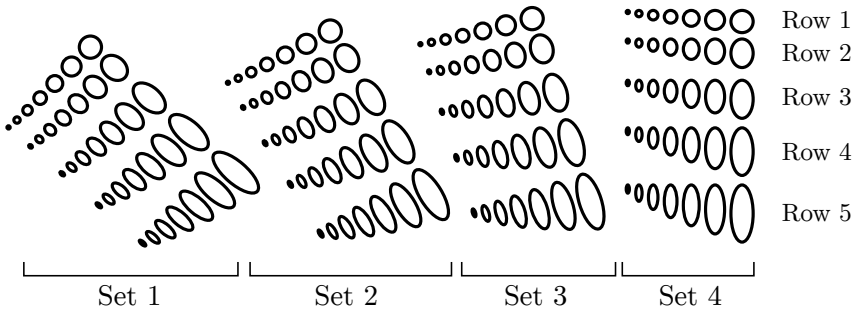


Figure 4.1: The set of ellipses used for the ellipse test.

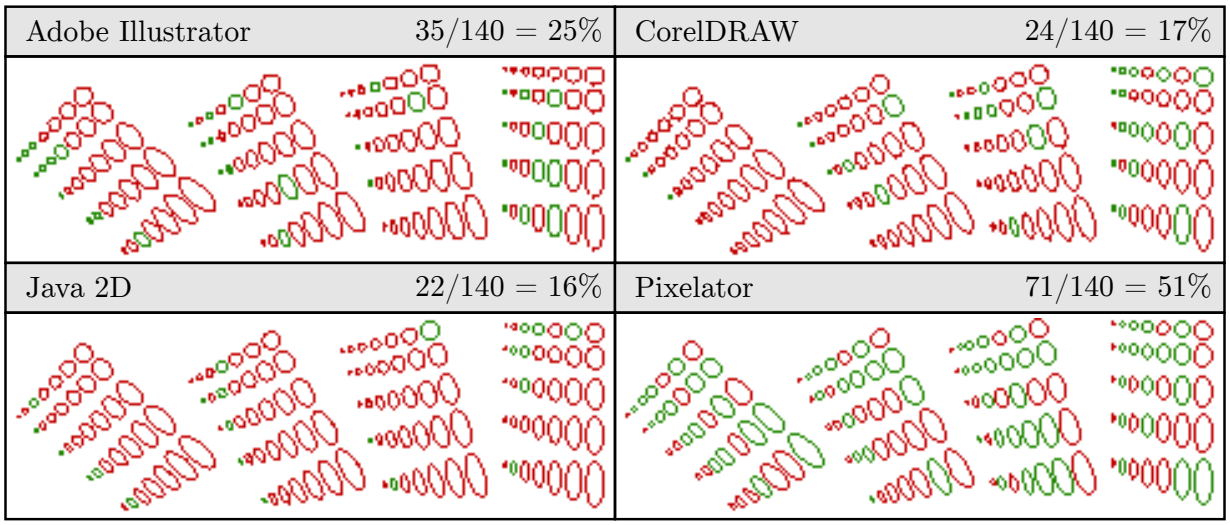


Figure 4.2: Ellipses under various affine transformations rasterized by Adobe Illustrator, CorelDRAW, Java 2D, and Pixelator. Symmetric shapes are in green and asymmetric shapes are in red. For each rasterizer, the percentage of green shapes is provided.

horizontal and vertical axes. Row 1 of each set contains circles of various sizes, each of which should be rasterized with the same width and height, and satisfy both 8-fold rotational and reflection symmetry. The rasterization results are shown in Figure 4.2. The green shapes are those that satisfy all the symmetry properties above, and the red shapes are those that do not.

In addition to satisfying these shape constraints, we also want the rasterized shapes to have as few artifacts as possible. The artifacts we look for include dropouts, blips, and L-shaped corners. Figure 4.3 shows three examples, each demonstrating one type of

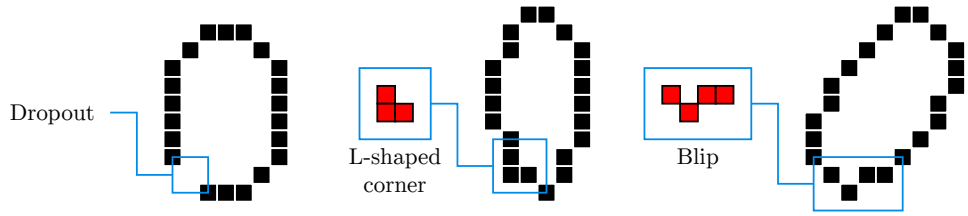


Figure 4.3: Artifacts to look for in the ellipse test: dropouts, L-shaped corners, and blips.

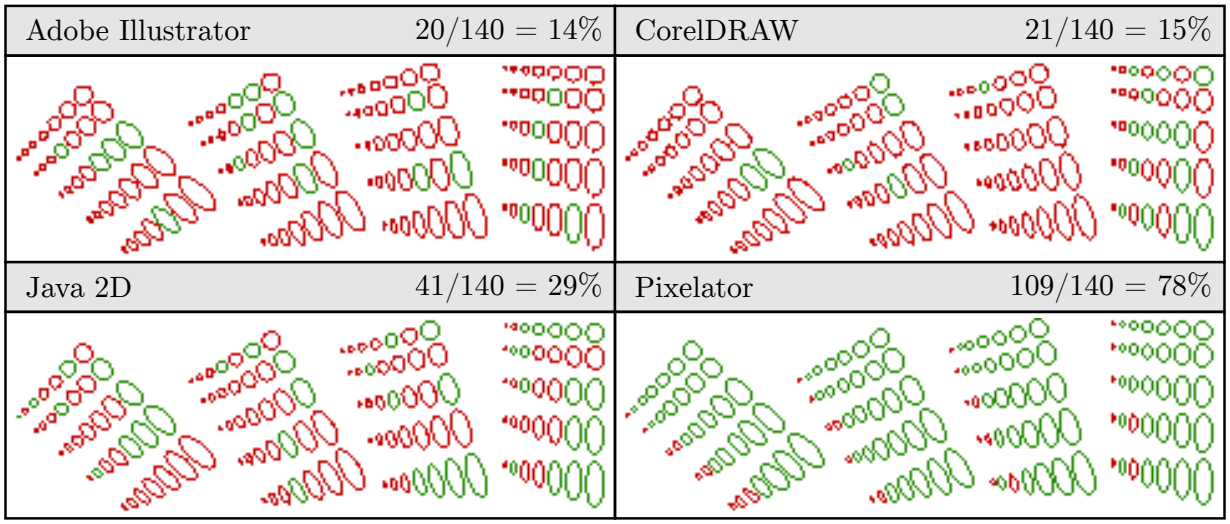


Figure 4.4: Ellipses under various affine transformations rasterized by Adobe Illustrator, CorelDRAW, Java 2D, and Pixelator. Shapes with blips, L-shaped corners, or dropouts are in green and the rest are in red. For each rasterizer, the percentage of green shapes is provided.

artifact. Figure 4.4 shows the result of the artifact search. If a shape contains any one of the three artifacts, then it is drawn in red. Otherwise, it is coloured green.

According to Figure 4.2, Illustrator slightly outperforms CorelDRAW and Java 2D, and Pixelator does a better job preserving symmetry than all three. However, nearly half of the ellipses rasterized by Pixelator are still asymmetric. As for the other artifacts, Figure 4.4 shows Java 2D outperforming Illustrator and CorelDRAW, and Pixelator performing better than all three.

Some existing rasterization algorithms do preserve symmetry, but they are designed for specific geometric primitives and are not generalizable. For example, Bresenham's circle-

and ellipse-drawing algorithms [26] can preserve symmetry when drawing axis-aligned circles and ellipses, but the techniques do not generalize to drawing rotated ellipses symmetrically.

Our goal is to develop an algorithm that preserves symmetry better than Pixelator without introducing additional artifacts. We will describe how Superpixelator preserves symmetry and other shape properties. For evaluation, we will apply the ellipse test to Superpixelator, and then compare its results to other rasterizers as well as a professional pixel artist.

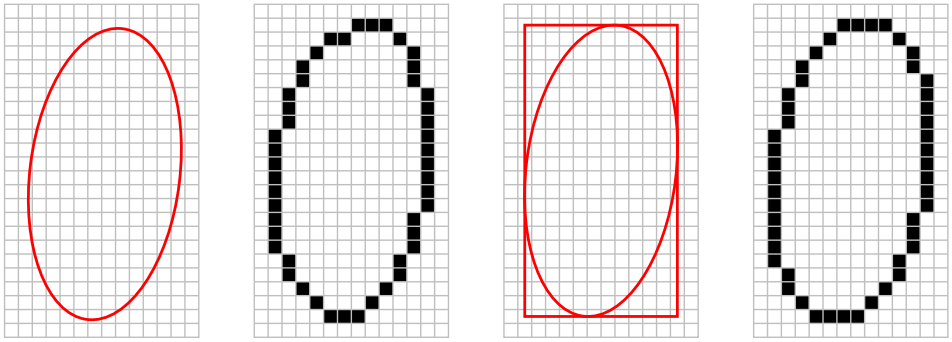
4.2 Algorithmic Approach

4.2.1 Bounding Box Corner Alignment

Pixelator does not preserve the overall symmetry of a path because it splits a path into curve segments and shifts them separately. For example, it would split the rotated ellipse in Figure 4.5a into four arcs and shift each one by its endpoints for better grid alignment, resulting in an asymmetric pixelation (see Figure 4.5b) even though the original vector path has 180° rotational symmetry about its centre.

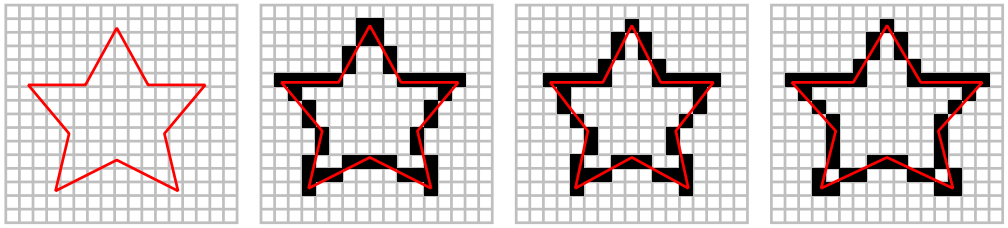
To preserve symmetry, let us examine the overall effect of shifting a path's curve segments. The shifting step moves all the critical points—including the maxima and minima in both the x- and the y-directions—to the closest pixel centres. The new bounding box of the shifted path therefore has corners aligned with pixel centres, as shown in Figure 4.5c. Knowing this, we can solve the asymmetry problem by first shifting the entire path to the new corner-aligned bounding box, and then ensuring that all subsequent steps are performed symmetrically about the box's centre. Operations such as rounding and computing the floor of a number should be made symmetric, and any point that lies on either the horizontal or vertical central axis should not be shifted. Figure 4.5d shows the symmetric pixelated ellipse created as a result of these changes.

If we know that a given shape is symmetric, we *could* simply process a single asymmetric portion of its outline and use transformed copies of its resulting pixels. The midpoint circle algorithm [26], for example, draws a circle using eight transformed copies of a 45° arc. However, we would rather not try to locate all possible symmetries from vector data because it is difficult and not easy to generalize to other shapes with different symmetries. Instead, it is easier to work with the alignment of its bounding box, with the assumption that a suitably tailored pixelation algorithm will then inherit the symmetries imposed on the bounding box relative to the pixel grid.



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

Figure 4.5: (a,b) Shifting curve segments causes paths to be drawn asymmetrically. (c,d) Aligning the corners of the bounding box with pixel centres helps preserve symmetry in the resulting pixelation.



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

Figure 4.6: (a) Pixelating a shape symmetrically is not easy. It can lead to (b) pixel clusters, or make the shape (c) too narrow or (d) too wide.

4.2.2 Preserving Other Global Properties

Symmetry is important, but there are other properties to consider as well when rasterizing a shape. Take the star-shaped path in Figure 4.6a, for instance. If we shift it to the nearest corner-aligned bounding box for symmetry preservation, the star will be centred horizontally between two columns of pixels (see Figure 4.6b), causing the topmost vertex to be drawn as a 2×2 pixel cluster.

We want to keep the acute angle looking sharp by representing it as a single pixel. We can translate the entire bounding box by half a pixel so that the topmost vertex lies on a pixel centre. If we decide to preserve symmetry, then the width of the translated star

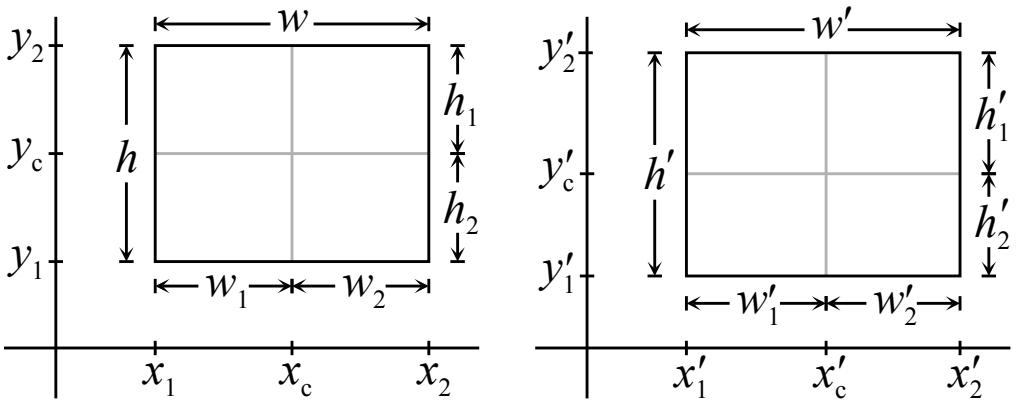


Figure 4.7: Bounding box corner alignment helps preserve symmetry and other properties of the vector shape after pixelation. The bounding boxes (a) before and (b) after the corner alignment as labelled as shown.

(outlined in red in Figures 4.6c and 4.6d) would be either 12 or 14 pixels. Either way, it would differ from the original 13-pixel-wide star by one pixel. This example demonstrates the difficulty in choosing the most important properties to preserve, whether they are symmetry, angle sharpness, or size.

To preserve shape properties, once again we look at the bounding box that contains the shape and how to modify it to achieve our goal. Figure 4.7a shows the parameters of such a bounding box. It ranges from x_1 to x_2 in the x-direction and from y_1 to y_2 in the y-direction. $x = x_c$ and $y = y_c$ are the vertical and horizontal bisectors. The w_i and h_i values are the various widths and heights as labelled in the figure.

Figure 4.7b shows the parameters of the bounding box *after* corner alignment. The corner alignment we discuss here is different from the corner alignment described in Section 4.2.1. For each corner, instead of simply moving the nearest pixel centre, we now consider all the pixel centres that are within distance 1 from it as possible positions to move to. These options give us several candidates for the new bounding box, and we wish to choose the one that preserves as many shape properties as possible.

We are interested in five shape properties that can be measured in terms of the bounding boxes: symmetry (C_s), dimension (C_d), position (C_p), acute angles (C_a), and aspect ratio (C_r). The symmetry property measures how similar the top and bottom halves of the path

are, as well as how similar the left and right halves are. It is defined as

$$C_s = \frac{\max(w_1', w_2')}{\min(w_1', w_2')} + \frac{\max(h_1', h_2')}{\min(h_1', h_2')}. \tag{4.1}$$

The dimension property refers to the change in width and height of the bounding box. It is given by

$$C_d = |w - w'| + |h - h'|. \tag{4.2}$$

The position property describes the change in absolute position of the bounding box. We measure the positional change for the sides of the bounding box as well as the bisectors. It is calculated as

$$\begin{aligned} C_p &= |x_1 - x_1'| + |x_c - x_c'| + |x_2 - x_2'| + & (4.3)\\ &\quad |y_1 - y_1'| + |y_c - y_c'| + |y_2 - y_2'|. & (4.4) \end{aligned}$$

The acute angle property refers to the number of acute angles that are aligned with bisectors that straddle two pixel rows or columns. We only consider these acute angles (as opposed to all the acute angles in the path) because they are the only ones that become 2×2 pixel clusters after bounding box adjustment. This value is given by

$$\begin{aligned} C_a &= \#\{\text{acute angles with } x = x_c \in \mathbb{Z}\} + & (4.5)\\ &\quad \#\{\text{acute angles with } y = y_c \in \mathbb{Z}\}. & (4.6) \end{aligned}$$

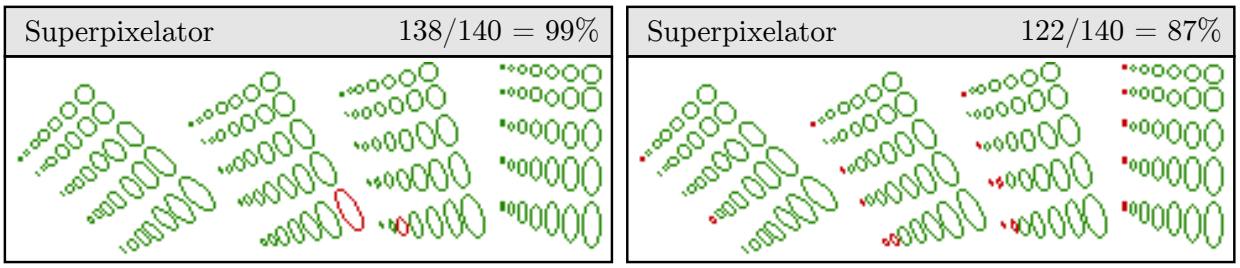
The last property is aspect ratio. Normally, an aspect ratio refers to the width-to-height ratio, but the dimension property already ensures that the width-to-height ratio would not change significantly. We are more concerned with shapes with equal width and height (such as circles or square) being stretch non-uniformly. The aspect ratio property measures this occurrence; it equals 1 if a shape with a square bounding box is stretched non-uniformly, and 0 otherwise. More precisely,

$$C_r = \mathbf{1}_{\{w=h, w' \neq h'\}}. \tag{4.7}$$

The cost function C_{total} we want to minimize is a weighted sum of these five costs:

$$C_{total} = k_s C_s + k_d C_d + k_p C_p + k_a C_a + k_r C_r. \tag{4.8}$$

Through trial and error, we arrived at the following set of weights: $k_s = 10000$, $k_d = 40$, $k_p = 1$, $k_a = 10$, and $k_r = 1$. These values indicate that symmetry is still considered most important but other factors are used as tie breakers.



(a) (b)

Figure 4.8: Ellipses under various affine transformations rasterized by Superpixelator. (a) Symmetric shapes are in green and asymmetric shapes are in red. (b) Artifact-free shapes are in green and shapes with dropouts, L-shaped corners or blips are in red. In both cases, the percentage of green shapes is provided in the top-right corner.

4.3 Evaluation

Our results were evaluated in two stages. First, we applied the previously used ellipse test to Superpixelator. Second, we conducted a qualitative evaluation to compare our results to drawings created by other rasterizers and a professional pixel artist. The results were analyzed by us as well as by two pixel artists. The artists also experimented with using our image editor which rasterizes using Superpixelator and provided feedback.

4.3.1 Ellipse Test for Superpixelator

We saw in the earlier ellipse test that Pixelator preserved symmetry 51% of the time while keeping 78% of the shapes artifact-free, and for both criteria, Pixelator significantly outperformed the other rasterizers. We applied the same test to Superpixelator and the results are shown in Figure 4.8. At 99%, Superpixelator preserved symmetry in nearly all the ellipses. Somewhat surprisingly, the number of artifact-free ellipses also increased by nearly 10% from Pixelator to Superpixelator. Drawing rotated curved shapes is a difficult task in pixel art and Superpixelator has successfully resolved this problem in the ellipse test.

4.3.2 Qualitative Evaluation Set-up

For the qualitative evaluation, we chose a set of 24 test shapes for a more comprehensive analysis on the rasterization quality of geometric primitives. All the shapes in the set are geometric primitives commonly found in both vector and raster image editors, including six rotated ellipses, six rotated rectangles, six rotated rounded rectangles, and six stars (from 3-pointed to 8-pointed). The shapes are deliberately not aligned with the pixel grid to make the pixelation task more challenging.

We rasterized these shapes using Superpixelator, Pixelator, CorelDRAW, Adobe Illustrator, and Java 2D. To compare these results to actual hand-drawn pixel art, we asked professional pixel artist Sven Ruthner[2] to draw all the shapes by hand as pixel art. Ruthner has been practicing pixel art professionally for a decade, and is well-known within the pixel art community. To get a sense of how much time is involved, Superpixelator took on average 450ms, Java 2D took about 70ms, and Ruthner spent about one hour drawing the 24 shapes.

The results are shown in Figure 4.9. To analyze these results, we took some drawing samples and compared them in terms of various artifacts and shape properties. In addition to our own analysis, we also wanted to get some expert opinion. For evaluation, we asked both Ruthner and eBoy[3]—an internationally renowned pixel art design firm specializing in detailed isometric cityscapes for advertisements—to provide feedback on the performance of our algorithm.

4.3.3 Qualitative Results Inspection

From Figure 4.9, it appears that Illustrator and CorelDRAW's results contain the most artifacts. To analyze the results more closely, we examine the remaining four sets of results by Java 2D, Pixelator, Superpixelator, and the pixel artist. We focus on three main criteria: path smoothness, symmetry, and angle sharpness. For each criterion, we will select four sample shapes that best illustrate the differences between the four result sets.

Figure 4.10 compares path smoothness. A smooth path should be drawn without blips and with as few jaggies as possible without deviating from the original vector path too much. Java 2D's result contains some blips and L-shaped corners; the rectangle is very jagged, to the point where its edges do not look straight at all. The other three sets of

[2] ptoing.blogspot.ca
[3] hello.eboy.com

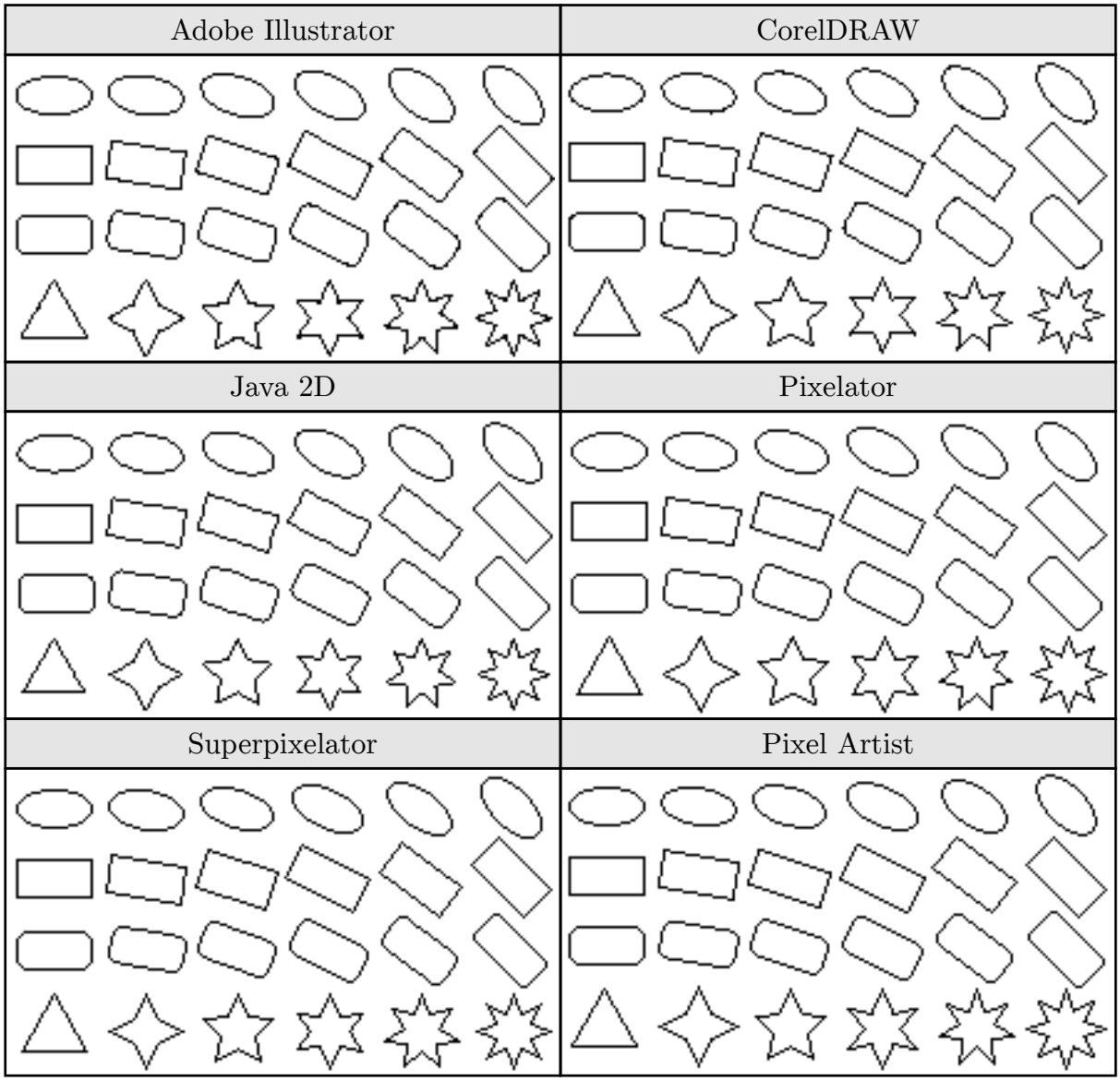


Figure 4.9: Vector shapes rasterized without antialiasing by Adobe Illustrator, CorelDRAW, Java 2D, Pixelator, Superpixelator, and Ruthner (pixel artist).

results all have sufficiently jaggie-free paths. The only minor problem is that Pixelator's ellipse looks slightly lopsided due to asymmetry.

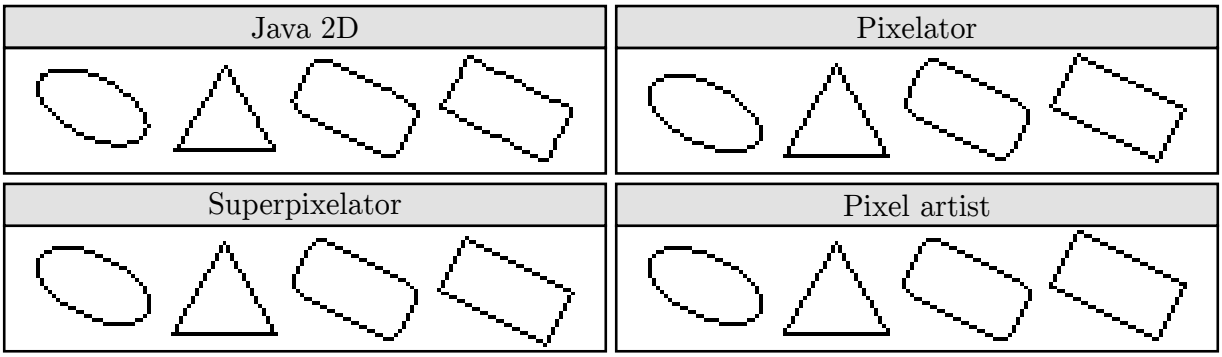


Figure 4.10: A comparison of path smoothness among rasterized results from Java 2D, Pixelator, Superpixelator, and Ruthner (pixel artist).

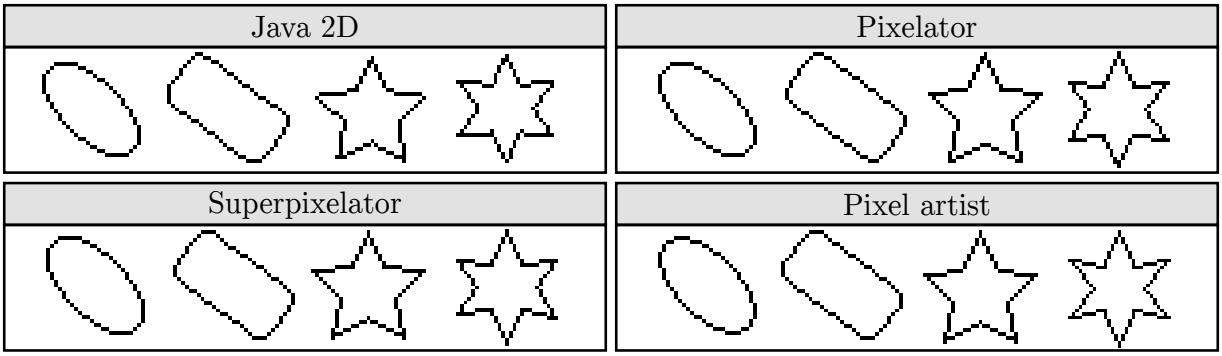


Figure 4.11: A comparison of shape symmetry among rasterized results from Java 2D, Pixelator, Superpixelator, and Ruthner (pixel artist).

Figure 4.11 focuses on symmetry. Java 2D preserved symmetry in the ellipse, but not in the other three shapes. Pixelator preserved symmetry in two of the four shapes (ellipse and the six-pointed star). Superpixelator and Ruthner both preserved symmetry quite well. The only difference is the six-pointed star; Superpixelator's result is slightly asymmetric whereas Ruthner's is perfectly symmetric. However, since preserving symmetry requires distorting the shape, Superpixelator's result looks more similar to a regular six-pointed star than Ruthner's.

Figure 4.12 compares how sharp angles are drawn. Java 2D did a poor job preserving sharp angles. Pixelator's results are better but the asymmetry sometimes causes problems (e.g., the top angle in the four-pointed star looks dull). Superpixelator and Ruthner both

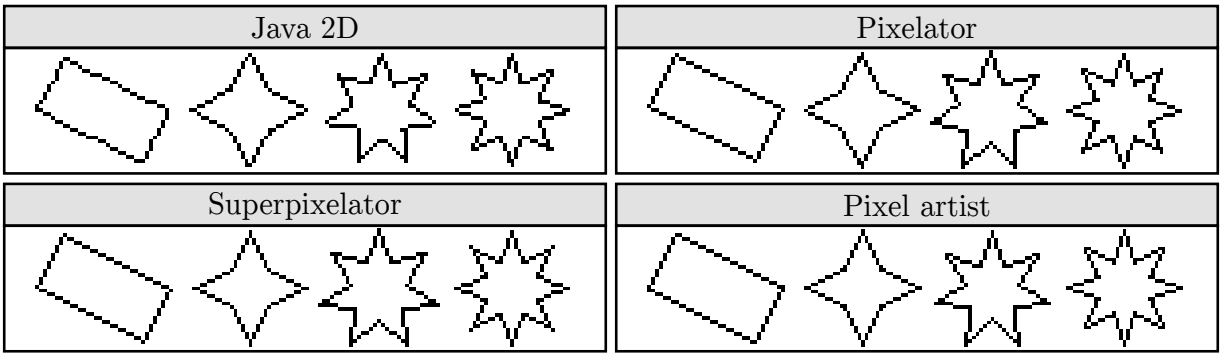


Figure 4.12: A comparison of sharp angles among rasterized results from Java 2D, Pixelator, Superpixelator, and Ruthner (pixel artist).

preserved angle sharpness quite well, even though they did not always draw angles the same way.

4.3.4 Artist Feedback

After analyzing the drawings, we also asked both pixel artists, Ruthner and eBoy, to provide their expert opinion. Both of them were asked to comment on the results in Figure 4.9. In addition, Ruthner experimented with our image editor, which uses Superpixelator as the built-in rasterizer. Figure 4.13 shows a screenshot of our editor.[4] It supports both vector and raster layers. Vector shapes drawn on a vector layer are pixelated immediately by Superpixelator, but remain editable as vector shapes. Raster layers can then be used to add pixel-level details to the final composition. The editor also has a split-pane view so that the user can compare the difference between Superpixelator and the Java 2D rasterizer.

When shown Figure 4.9, both artists agreed that the quality of Superpixelator's drawings surpasses all other rasterizers, and matches that of hand-drawn images. Ruthner commented that the straight lines could be drawn with more regularity in the pixel patterns. Figure 4.14 shows how the pixel artist drew lines by repeating pixel patterns to make them look straighter, whereas Superpixelator did not. In Chapter 6, we will describe a line-drawing algorithm that specifically deals with this issue.

As for our image editor, Ruthner felt that the user interface can be improved since it does not have as many functions as a typical pixel art editor. Also, he commented

[4]It can be downloaded from our project webpage: https://sites.google.com/site/tiffanycinglis/research/rasterizing-and-antialiasing-vector-line-art-in-the-pixel-art-style

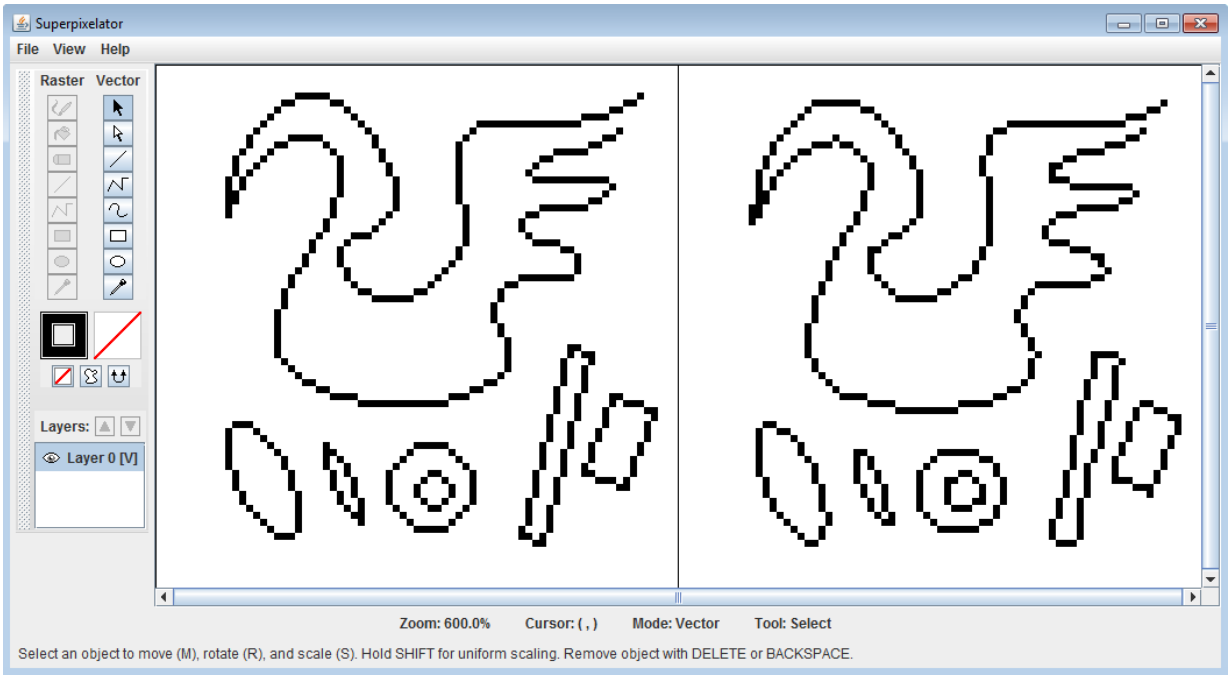


Figure 4.13: A screenshot of our drawing tool. The left pane shows rasterization by Superpixelator while the right pane contains rasterization by Java 2D.

that when editing a vector path, the automatic pixelation feels somewhat jerky due to the grid alignment step in our algorithm. Currently we do not have a good solution for this problem; however, since our results look much better than the default Java 2D rasterizer, we believe it is a fair trade-off.

4.4 Summary

Superpixelator improves upon Pixelator by taking shape-level considerations in account. Specifically, the Superpixelator algorithm tries to compromise between various shape-level properties such as symmetry, size, position, sharp angles, and aspect ratio. These changes greatly improve the drawing of geometric primitives at low resolutions.

The evaluation consists of comparing our results to those of a pixel artist and other rasterizers. Our analysis shows that, according to several professional pixel artists, our results are more visually appealing than all other rasterizers and similar to the hand-drawn images by the artist. In the next chapter, we will augment Superpixelator with

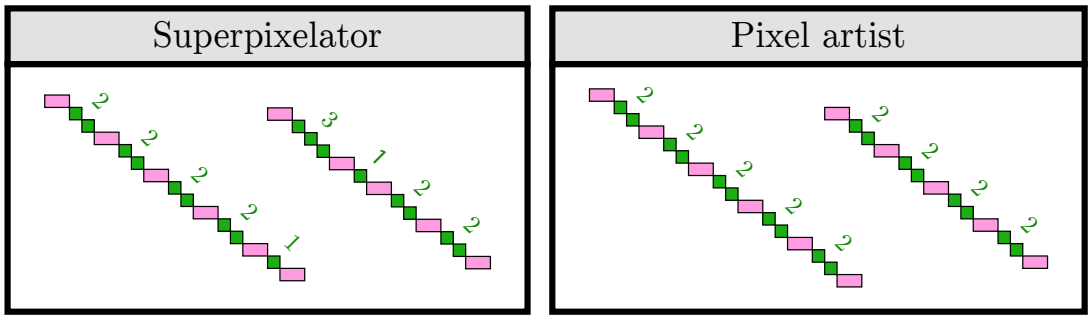


Figure 4.14: The pixel artist uses repeating pixel patterns when drawing a straight line, while Superpixelator does not. The numbers indicate how long the runs of length-1 spans are.

manual antialiasing, and evaluate it in the same way. In Chapter 6, we will return to the problem of drawing straight lines with more regular pixel patterns.

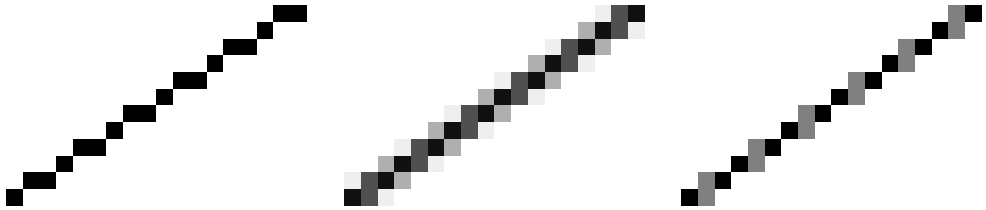
Chapter 5

Manual Antialiasing

Some pixel-level limitations are difficult to overcome. For example, an imperfect line with slope $^2/_3$ cannot be drawn without jaggies (see Figure 5.1a). One common way to mitigate the impact of these artifacts is to use antialiasing. Pixel artists often talk about two types of antialiasing: *automatic* and *manual*. Automatic antialiasing refers to the algorithms used by image editors when rasterizing a vector image, while manual antialiasing is what pixel artists do by hand.

There are several differences between automatic and manual antialiasing. Automatic antialiasing does not limit the colour palette used and at low resolutions the result can often look blurry (see Figure 5.1b). In contrast, manual antialiasing is done by hand using a limited palette (see Figure 5.1c). When applied to line art, manual antialiasing creates a cleaner look because the lines look thinner and sharper. The fact that manual antialiasing avoids blurring the image is important because in pixel art, due to the low resolution, fattening a line even by a row of pixels on either side wastes too many pixels that could be used for other purposes such as shading and adding texture.

The goals of manual antialiasing are similar to those of low-resolution font rasterization with antialiasing. Both methods seek to preserve details and create smoother-looking outputs without sacrificing clarity and readability. Font rasterization uses antialiasing to improve accuracy and font hinting to increase contrast, thereby improving legibility at low resolutions [59]. In this chapter, we describe a manual antialiasing algorithm that antialiases vector shapes at low resolutions while preserving clarity.



(a) (b) (c)

Figure 5.1: A line of slope $^2/_3$ drawn (a) without antialiasing, (b) with automatic antialiasing, and (c) with manual antialiasing.

5.1 Antialiasing Algorithms

Automatic antialiasing is usually an approximation of the pixel coverage of a vector image using pixels with varying opacity levels, and can be computed with many different antialiasing algorithms, including supersampling, multisampling [3], and fast approximate antialiasing [43]. The problem is that, even if we can compute the exact pixel coverage, the result would be mathematically correct, but still not necessarily what one would want in pixel art. In a pixel art, we want to apply antialiasing using a small set of colours, and smaller images should use less antialiasing to avoid looking too blurry.

Manual antialiasing is what pixel artists do by hand to soften the transitions between pixel spans. Since it is difficult to develop an intuition for this technique, many serious pixel artists follow some rules when applying manual antialiasing. Ruthner's tutorial[1] discusses a method related to pixel coverage, which is similar to many automatic antialiasing algorithms. Gowers's tutorial[2] relies on the idea of adding up pixel opacities to determine the apparent thickness. As with many art tutorials, these methods are not full algorithms, but rather suggestions on how to approach the problem and how to evaluate the outputs. Also, both artists demonstrated their methods using only simple shapes, whereas we are more interested in applying manual antialiasing to general shapes.

5.2 Algorithmic Approach

In this chapter, we augment Superpixelator by adding a manual antialiasing algorithm. Basically, it mimics what pixel artists do by drawing paths with less blur using a limited

[1] http://storage5.static.itmages.ru/i/11/0912/h_1315823707_4054244_c0e6ef162c.png
[2] neota.castleparadox.com/aa_tutorial.html

palette. The beginning of the algorithm is the same as before, but then it diverges part way through to allow for either aliased or antialiased pixelation.

Let us assume that we are given a black path of unit thickness, and are asked to pixelate it on a white background using four shades of grey. We first preprocess the path by adjusting the bounding box, splitting it up and grid-aligning each subpath, as described in Chapter 4. Then manual antialiasing is applied in four steps. In the discussion that follows, we speak interchangeably of the grey level of a pixel and its opacity.

First we select parts of the path that do *not* require antialiasing. In the pixel art examples we have analyzed, lines that are horizontal, vertical, or of slope ± 1 are almost never antialiased. Therefore we do the same in our algorithm.

Next, we draw the remainder of the path with antialiasing. To avoid blurriness, we first rasterize a thinner version of each curve via supersampling. For example, Figure 5.2a shows a curve rasterized with thickness 0.5; the numbers in the figure represent pixel opacity values in the range $[0, 100]$. Notice that the resulting pixelated path has low opacity, and it cannot be fixed simply by doubling the opacities because it would make the overall line thickness look uneven. We need some way of normalizing the opacities so that the pixelated path looks one pixel thick everywhere.

Our approach to normalization is based on the idea of apparent thickness described in Gowers's antialiasing tutorial. In a pixelated path, we can pick a pixel, calculate the total opacity in that row and column, and whichever value is less, divide it by 100 to get the apparent thickness at that pixel. The reason the pixelated path in Figure 5.2a looks thinner than one pixel is because its apparent thickness is less than 1 everywhere. Gowers only experimented with straight lines in his tutorial, but our algorithm will apply this idea more generally to paths that are monotonic and slope-monotonic. Even though this measure of apparent thickness may not be exactly how our eyes interpret a pixelated image, it works quite well for the purpose of drawing pixel art, and we use it to adjust path thickness during antialiasing.

Normalizing the apparent thickness at every pixel in a pixelated path requires solving an optimization problem, which is too time-consuming. Instead, we use an approximate method that locally scales the opacity of each pixel. To normalize the path thickness in Figure 5.2a, let o_{ij} be the original opacity for the pixel of row i and column j. Then for each pixel, we scale its opacity with

$$p_{ij} \quad = \quad \frac{t}{\min\left(\sum_i o_{ij}, \sum_j o_{ij}\right)} p_{ij}, \tag{5.1}$$

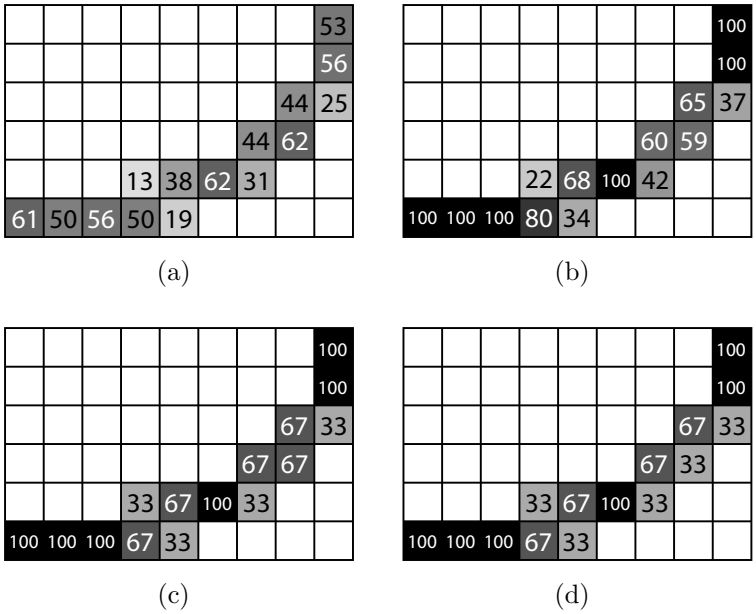


Figure 5.2: We apply manual antialiasing by (a) rasterizing a thinner path, (b) normalizing opacities, (c) reducing colour count, and (d) renormalizing opacities.

where t is the vector path thickness and p_{ij} is the new opacity. Figure 5.2b shows the normalized path.

To reduce the colour count, we replace the colour of each pixel by its closest match in a given palette. Figure 5.2c shows the resulting path. To ensure the path thickness has not changed too much, we apply normalization again, allowing pixels to be replaced only by colours in the limited palette. Figure 5.2d shows the final pixelated path.

Different results are produced depending on how thin we choose to rasterize the path initially. The thinner the path, the less antialiasing will be applied, which means the path will look sharper but also more jagged. Based on samples of pixel art with antialiasing, we found 0.75 to be a good thickness for the initial rasterization.

If a path contains line segments that are horizontal, vertical, or of slope ± 1, then those segments will be pixelated without antialiasing. As a result, the pixelated path will look uneven in thickness where it transitions from being antialiased to aliased. In this case, we apply the initial rasterization at 0.5 thickness so that the transition is less noticeable.

5.3 Qualitative Evaluation

To evaluate the manual antialiasing algorithm that we added to Superpixelator, we designed a qualitative analysis using a similar experimental set-up as the one described in Section 4.3.2. We gave the same 24 shapes to pixel artist Ruthner and asked him to redraw them as antialiased pixelated shapes. Since pixel artists often use a small palette for antialiasing, we asked him to use a greyscale palette consisting of white (black at 0% opacity), light grey (25% opacity), grey (50% opacity), dark grey (75% opacity), and black (100% opacity).

We compared Superpixelator to the pixel artist as well as CorelDRAW Graphics Suite X6, Adobe Illustrator CS5.1, and Java 2D. The results are shown in Figure 5.3. Note that Pixelator is not included because it does not support antialiasing. For rasterizers that support antialiasing, we want to create a fair comparison by keeping the number of colours the same. Since Superpixelator uses five fixed shades of grey, we apply nearest-neighbour palette reduction to the other antialiased results so that they use the same palette.

Drawing the 24 shapes with antialiasing took Superpixelator about 950ms and Java 2D about 70ms. Ruthner reported spending approximately seven hours, which includes the initial drawing time and some later revisions. Overall, pixelating with antialiasing is orders of magnitude more time-consuming for the artist than for the antialiasing algorithms.

5.4 Qualitative Results Inspection

To analyze the results more closely, we chose to focus on certain shapes rasterized by Java 2D, Superpixelator, and the pixel artist. Our evaluation is based on two criteria: blurriness and apparent path thickness.

Figure 5.4 compares the blurriness of the antialiased shapes. Blurriness can be roughly defined as the number of pixels used to draw a shape. For all four shapes, Java 2D's drawings are the most blurry. Even for the axis-aligned rectangle and rounded rectangle, for which we expect all four sides to be drawn as one-pixel-thick black lines, Java 2D drew them as two-pixel-thick grey lines. In contrast, both Superpixelator and the pixel artist were able to draw the two shapes in a clean way.

For the rotated rectangle, because its edges have slope ± 1, both our algorithm and Ruthner drew them without any antialiasing, while Java 2D applied antialiasing and made all the edges blurry. Finally, for the four-pointed star, the results do not look too different at first glance. However, a closer look reveals that Superpixelator used fewer pixels in its

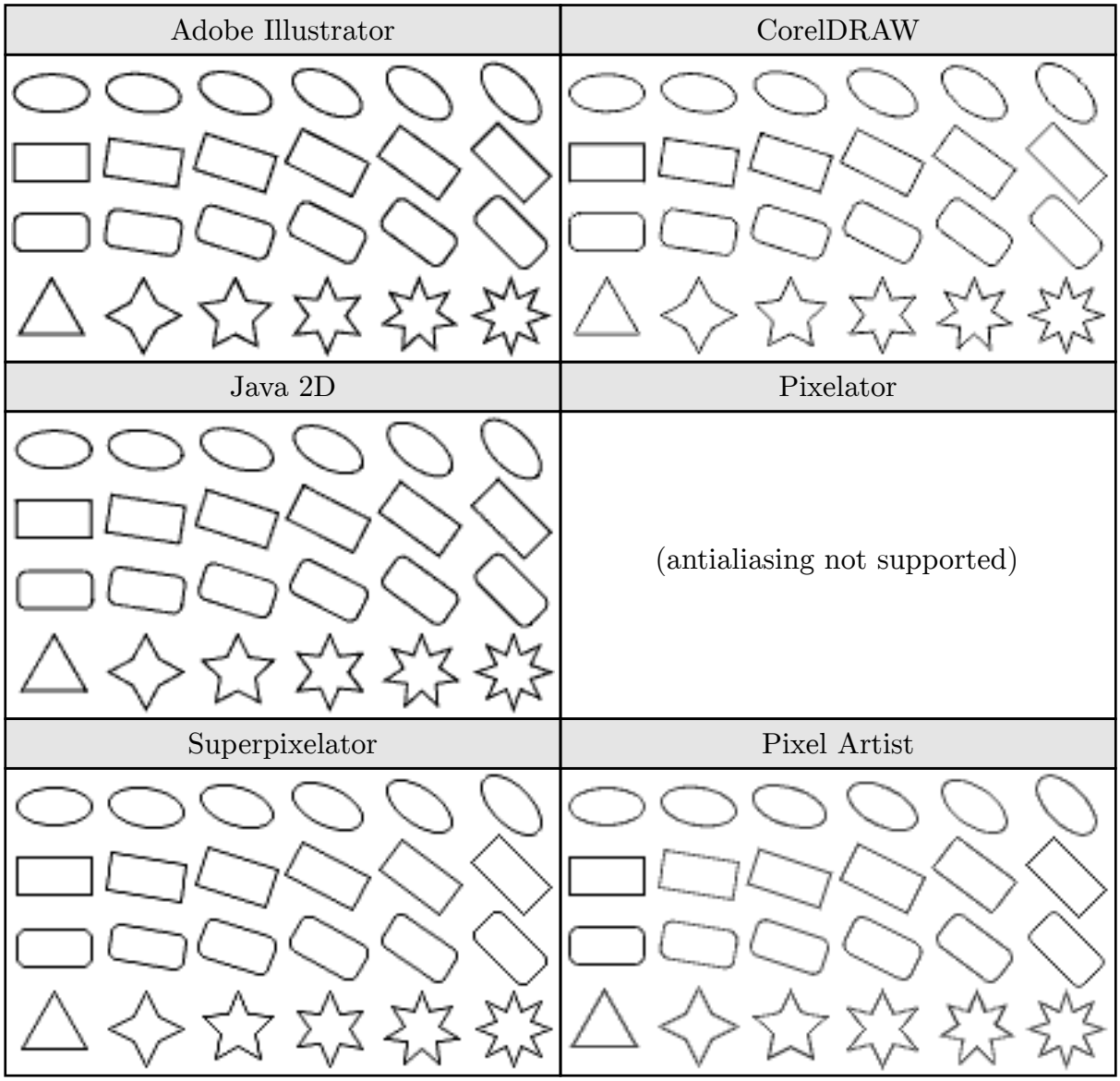


Figure 5.3: Vector shapes rasterized with antialiasing by Adobe Illustrator, CorelDRAW, Java 2D, Superpixelator, and Ruthner (pixel artist).

drawing than both Java 2D and the pixel artist. Ruthner used lighter shades of grey and maintained symmetry while Java 2D used dark shades and did not preserve symmetry.

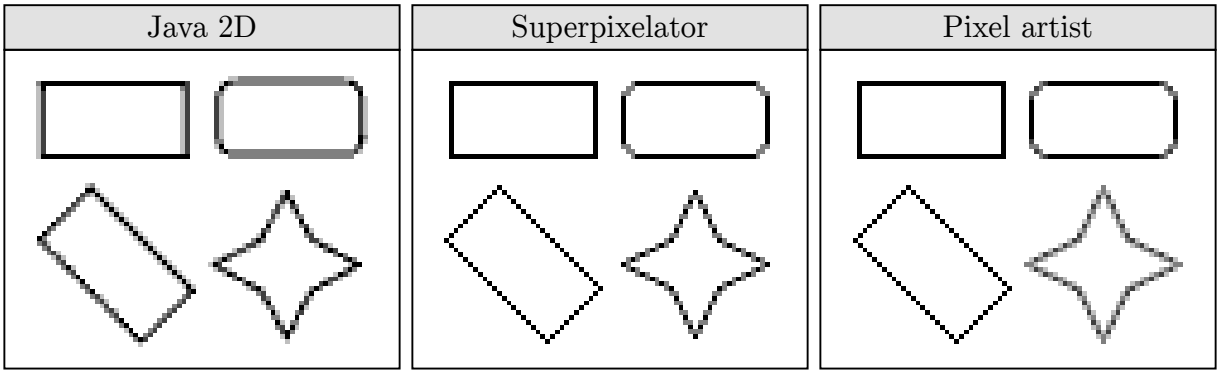


Figure 5.4: A comparison of blurriness among antialiased results from Java 2D, Superpixelator, and Ruthner (pixel artist).

It is interesting to note that for the 4-pointed star, Superpixelator chose to use only three of the five available shades of grey while Ruthner and Java 2D used all five shades. This decision by Superpixelator is a consequence of the first step of the algorithm in which we rasterized a thinner path. The thinner this path is, the fewer colours the output image potentially uses. In Chapter 6, we will describe another antialiasing algorithm for lines that allows for more flexibility in choosing the number of colours to use.

The next criterion to analyze is the apparent path thickness, as illustrated in Figure 5.5. As described earlier, the apparent thickness of a path is defined in terms of pixel opacities. We noticed that, other than axis-aligned shapes that can be drawn without antialiasing, most of the shapes drawn by Ruthner use noticeably lighter shades of grey; this was unexpected because it makes the paths look thinner than they should be and the images look washed out.

To understand why the pixel artist did not simply use darker pixels, we recoloured each pixel one shade darker in the given palette, as shown in Figure 5.6. The recolouring not only did not improve the drawing, it made much worse because it looks too dark and even more blurry than Java 2D's results. This experiment suggests that the reason Ruthner used lighter pixels is to avoid blurriness. Some of the shapes (e.g., near-horizontal or near-vertical lines) require a lot of antialiasing to look smooth, and using lighter pixels can hide the blurriness due to the pixels used for antialiasing.

We asked Ruthner to comment on the results. He believes that our algorithm once again outperforms the other rasterizers. Even though there are differences between Superpixelator's results and his (such as the amount of antialiasing applied and the shades of

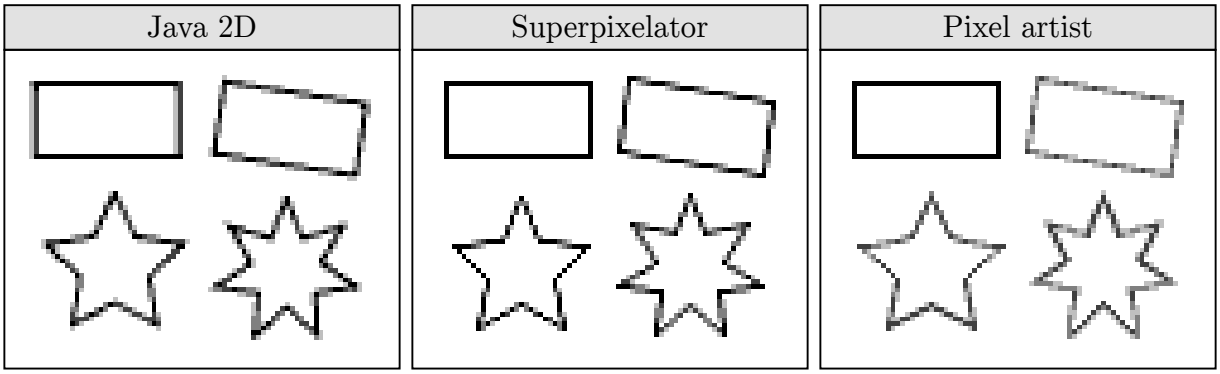


Figure 5.5: A comparison of apparent path thickness among antialiased results from Java 2D, Superpixelator, and Ruthner (pixel artist).

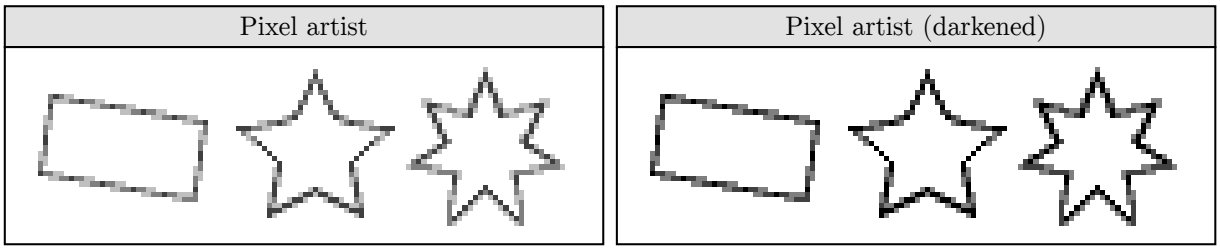


Figure 5.6: The shapes drawn by the pixel artist (left) contain relatively light-coloured pixels. We darkened them (right) for comparison.

grey used), he considers them to be stylistic differences and believes that our results are good compared to pixel artists in general.

As before, Ruthner pointed out that Superpixelator's rasterization of straight lines lacks regularity in pixel patterns, which he considers to be an important attribute in pixel art. Figure 5.7 shows close-ups of some of the straight lines drawn by Superpixelator and by Ruthner. We identify repeating pixel patterns by drawing boxes around them. Within each line, different colours correspond to different patterns. While Ruthner always drew each line by tiling the same pattern, Superpixelator used up to three different patterns per line.

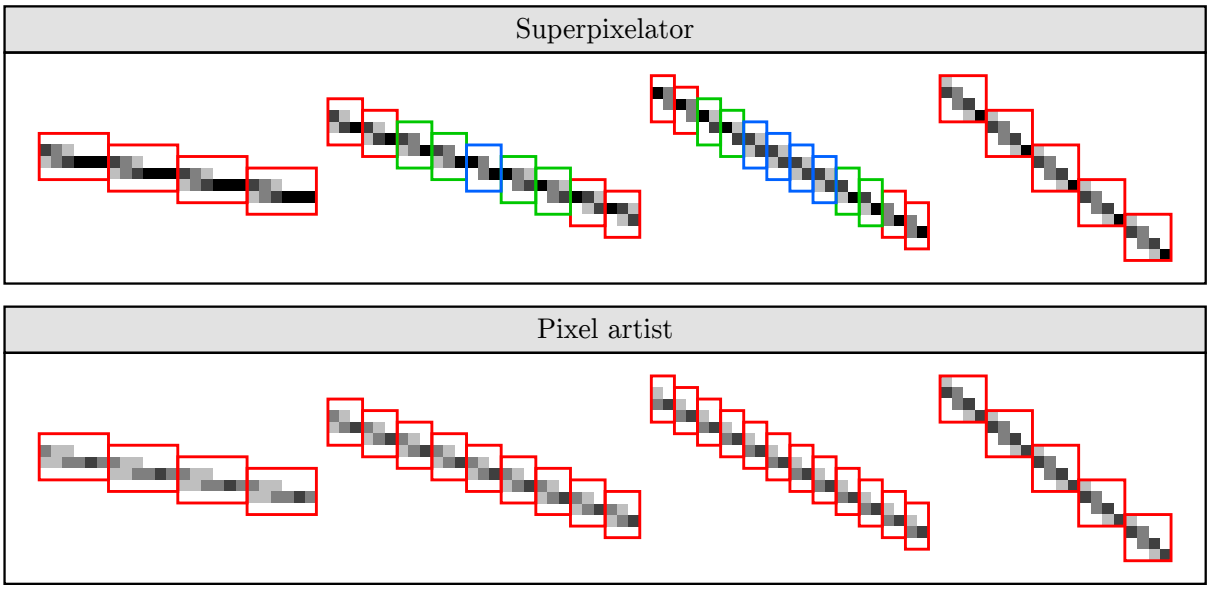


Figure 5.7: To analyze the regularity of pixel patterns in the antialiased shapes, we identified repeating units in both Superpixelator and the pixel artist's results, and drew boxes around them such that, for each line, the boxes of the same colour contain the same pixel units. The pixel artist's results are more regular than that of Superpixelator.

5.5 Summary

We added a manual antialiasing algorithm to Superpixelator so that the pixel art produced can be antialiased in a clean, non-blurry way using a limited palette. We tested the algorithm against other rasterizers and a professional pixel artist. The results with antialiasing are similar to the results without antialiasing that we discovered in Chapter 4: namely, Superpixelator outperforms other rasterizers and is on par with the pixel artist's drawings. The only difference is that the artist uses more regular pixel patterns to draw straight lines while Superpixelator does not. In the next chapter, we will describe a line-drawing algorithm that directly addresses this difference.

Chapter 6

Drawing Straight Lines

From the user study and qualitative evaluation in Chapters 4 and 5, we observed that when drawing either aliased or antialiased lines, pixel artists tend to create highly regular pixel patterns. Rasterization algorithms such as Bresenham's and Wu's sometimes produce such pixel patterns as well, but as by-products of the error minimization rather than deliberate aesthetic goals.

For us, regularity in pixel patterns is important for several reasons. First, it is what pixel artists do. To mimic the hand-drawn pixel art style, we need to understand how to generate these pattern algorithmically. One may argue that pixel artists use regular pixel patterns because humans find it easier to work with patterns, and that algorithms need not heed such constraints. While this is true, we believe that the pixel pattern regularity is also a good measure of how suitable a line is for pixel art.

Small changes can significantly affect the amount of regularity in a line. Figure 6.1 shows two aliased lines rasterized by Bresenham's algorithm and two antialiased lines rasterized by Wu's algorithm. The aliased line in Figure 6.1a contains a jaggie but by increasing its height by one, we can remove the jaggie to get a perfect line of slope 1 as shown in Figure 6.1b. The antialiased line in Figure 6.1c contains no obvious patterns, but by reducing its height by one, it can be redrawn with much higher regularity using only two colours, as shown in Figure 6.1d. For simple cases, pixel artists may be able to determine through trial and error what adjustments are needed to increase the amount of regularity. However, it would require an algorithm to detect and fix more difficult cases. Our goal is to create such an algorithm.

In addition to using pattern regularity to categorize lines, we may also be able to take advantage of their repetitive nature to reduce computation time. In other words, if we

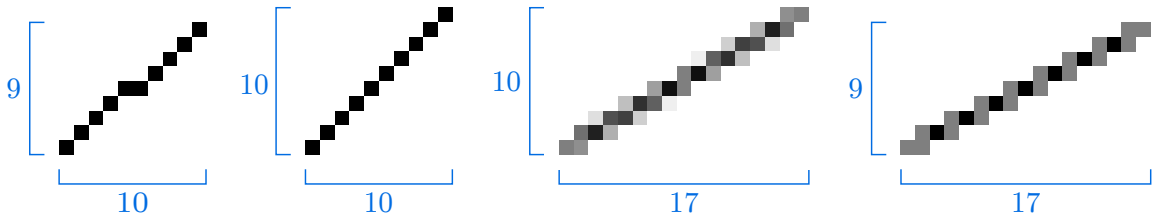


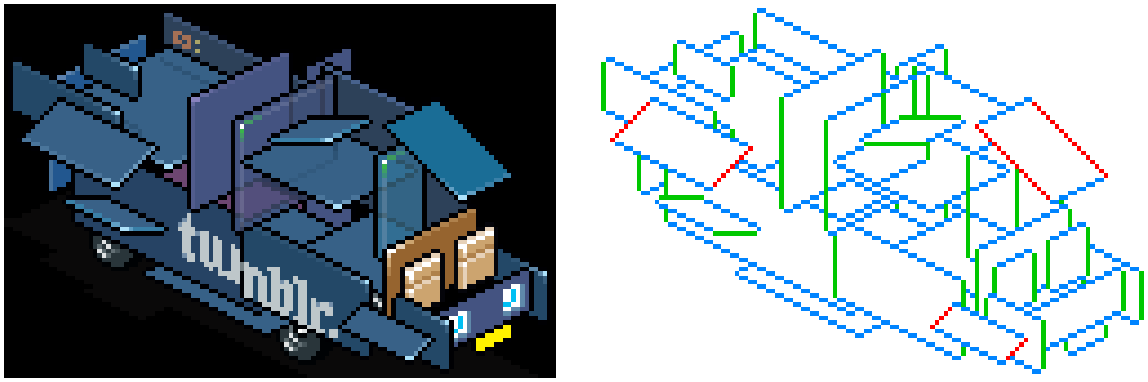
Figure 6.1: (a) A 10×9 aliased line is drawn with less regularity than (b) a 10×10 aliased line. (c) A 17×10 antialiased line is drawn with less regularity than (b) a 17×9 antialiased line. The aliased lines are drawn with Bresenham's algorithm [27] and the antialiased lines are drawn with Wu's algorithm [60].

can determine the amount of repetition in a pixelated line, then not only can we provide a more compact representation of the line, but it can also be drawn in less time.

Being able to draw straight lines well would benefit isometric pixel artists the most. In isometric pixel art, it is common to modify designs significantly to fit the pixel grid. For example, Figure 6.2a shows an isometric pixel truck designed by eBoy. In Figure 6.2b, we extracted the line art, kept only the straight lines, and coloured them based on their pixel patterns. The green lines are either horizontal or vertical, the red lines consist of lines of slope ± 1, and the blue lines consist of lines of slope ± 2 and $\pm \frac{1}{2}$. Notice that all the lines used have perfect slopes (i.e., integers or integer reciprocals), suggesting that the line slopes were deliberately chosen. Even the imperfect lines used in isometric pixel art tend to have highly regular pixel patterns. If we can identify which lines are "good" and which ones are "bad", then we can help artists design and draw isometric pixel art more effectively and efficiently.

In this chapter, we describe a line-drawing algorithm called the Euclidean Line-Drawing Algorithm (ELDA). As the name suggests, this algorithm uses the Euclidean algorithm as a subroutine for drawing lines. The algorithm has two flavours: ELDA-A for aliased rasterization and ELDA-AA for antialiased rasterization. The goal of ELDA is to draw lines at low resolutions with high pixel regularity as pixel artists would.

We will start by discussing similarities and differences between aliased and antialiased lines, and then describe the algorithms in greater detail. Note that, for antialiasing, we consider only the problem of drawing lines up to one pixel thick, because our method controls the *apparent* line thickness by manipulating pixel opacities; drawing thicker lines



(a) (b)

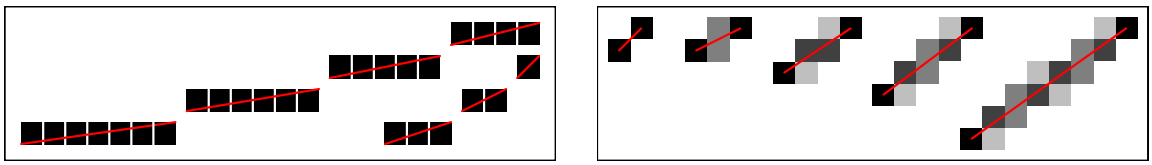
Figure 6.2: (a) An isometric truck by eBoy, and (b) its skeletal structure consisting of only perfect lines. The green lines are either horizontal or vertical, the red lines consist of lines of slope ± 1, and the blue lines consist of lines of slope ± 2 and $\pm\frac{1}{2}$. Used with permission.

effectively may require changing the *actual* line thickess, which is beyond the scope of this work.

6.1 Aliased vs. Antialiased Lines

Since our main goal is to augment the regularity of both aliased and antialiased lines, we will start by defining *units*, which are short pixel patterns used to construct a longer line. An *aliased unit* is simply a pixel span since an aliased line consists of several pixel spans joined at their diagonal corners. Several examples of aliased units are shown in Figure 6.3a. Within each unit, we have drawn a red line connecting opposite corners. In Chapter 3, we conjectured that, when spans are used in a pixelated line, they are perceived as diagonal line segments that join to form a polygonal path approximation of the line. For this reason, we sometimes refer to a vertical aliased unit of length n as a unit of slope $\pm n$, and a horizontal aliased unit of length n as a unit of slope $\pm\frac{1}{n}$.

An *antialiased unit* also represents a diagonal line but, unlike aliased units, its width and height can be any nonzero integers. Figure 6.3b shows examples of antialiased units representing, from left to right, line segments of slope 1, $\frac{1}{2}$, $\frac{2}{3}$, $\frac{3}{4}$, and $\frac{5}{7}$ (the line segments are drawn in red).



(a) (b)

Figure 6.3: (a) Aliased units join opposite pixel corners, while (b) antialiased units join opposite pixel centres.

One significant difference between aliased and antialiased units is how their lengths are defined. In Figure 6.3, we see that the red lines in aliased units join pixel corner to pixel corner, while those in antialiased units join pixel centre to pixel centre. Therefore, an antialiased unit that is 5 pixels wide and 4 pixels tall actually has a width of 4 and a height of 3, like the line segment it represents. When we refer to the dimensions of aliased and antialiased lines, we also consider them to be measured corner-to-corner or centre-to-centre, respectively.

The reason for measuring aliased lines corner-to-corner is to allow perfect pixelated lines to be drawn. Bresenham's algorithm measures lines centre-to-centre and it draws lines by iteratively choosing pixels that minimize the error (the error of a pixel is proportional to the distance from its centre to the line). The problem with this approach is that it cannot draw most perfect lines. For example, Figure 6.4a shows the line we want to draw: a perfect line with slope $\frac{1}{3}$ and length 21. However, no matter what the input vector line is—whether its slope is exactly or near $\frac{1}{3}$—Bresenham's algorithm cannot draw a perfect pixelated line of slope $\frac{1}{3}$ because it does not consider this pixel configuration to be error-minimizing. Figure 6.4b shows another example of a perfect line with slope $\frac{1}{4}$ and length 24 that Bresenham's algorithm cannot draw. To fix this problem, we need to measure lines corner-to-corner so that perfect lines would be error-minimizing solutions. To summarize, our algorithm will measure aliased lines and units corner-to-corner, and antialiased lines and units centre-to-centre.

Figure 6.5a shows three antialiased lines of slope $\frac{11}{15}$. When the pixels are sufficiently small, the three lines appear visually similar. However, the antialiasing is done in three different ways using different combinations of antialiased units. Each line uses two different antialiased units, and indicated by the red and blue boxes in Figure 6.5b. In Figure 6.5c, the pixelated lines are removed, leaving only the boxes. The boxes' diagonals form a polygonal path that approximates the original line. The shorter the units, the less accurate the overall approximation is, as evident from the increasing jaggedness of the polygonal path from left

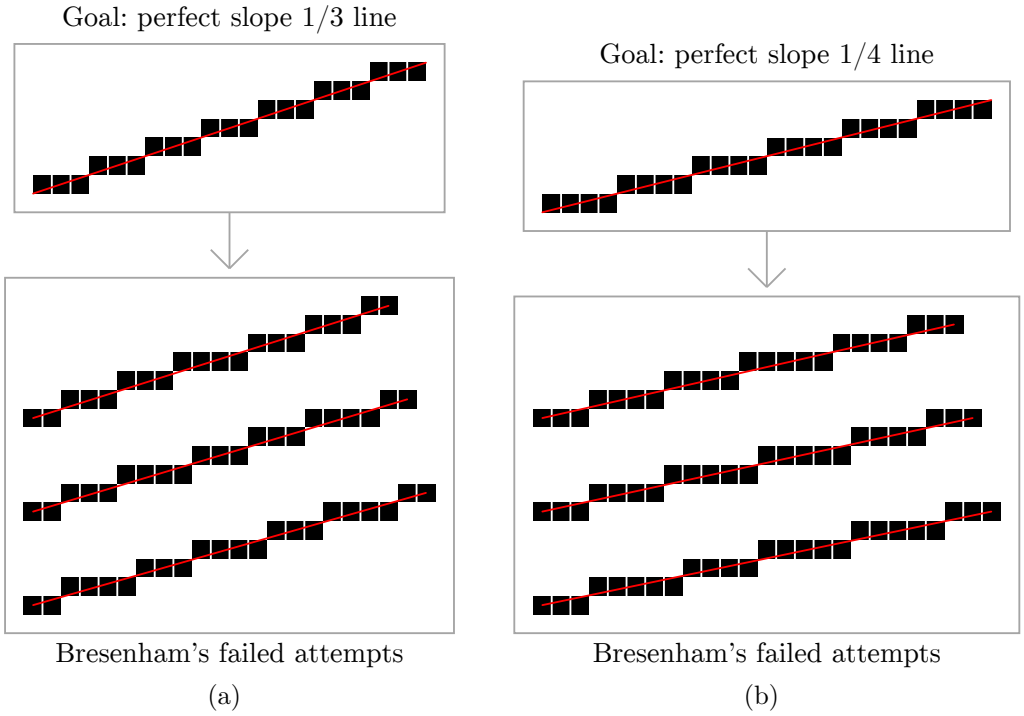


Figure 6.4: Bresenham's algorithm cannot draw either (a) a perfect line of slope $\frac{1}{3}$ or (b) a perfect line of slope $\frac{1}{4}$ because it measures lines centre-to-centre.

to right. On the other hand, shorter units use fewer colours, and provide more regularity to the overall pixel structure. Therefore, there is a trade-off between using longer units and shorter units.

6.2 Euclidean Line-Drawing Algorithm - Aliased

The aliased version of our algorithm, ELDA-A, determines which units are needed to construct a pixelated line and how to arrange those units. We will start by describing the main idea of the algorithm using an example, and then discuss the steps in detail.

Given a line joining two pixel centres, its slope is always a rational number. Suppose we want to draw a line with a width of 22 and a height of 28. Since the slope $\frac{28}{22}$ is between 1 and 2, the pixelated line can be constructed out of vertical units of lengths 1 and 2. Let the number of length-1 units and length-2 units be A and B, respectively. Then, by

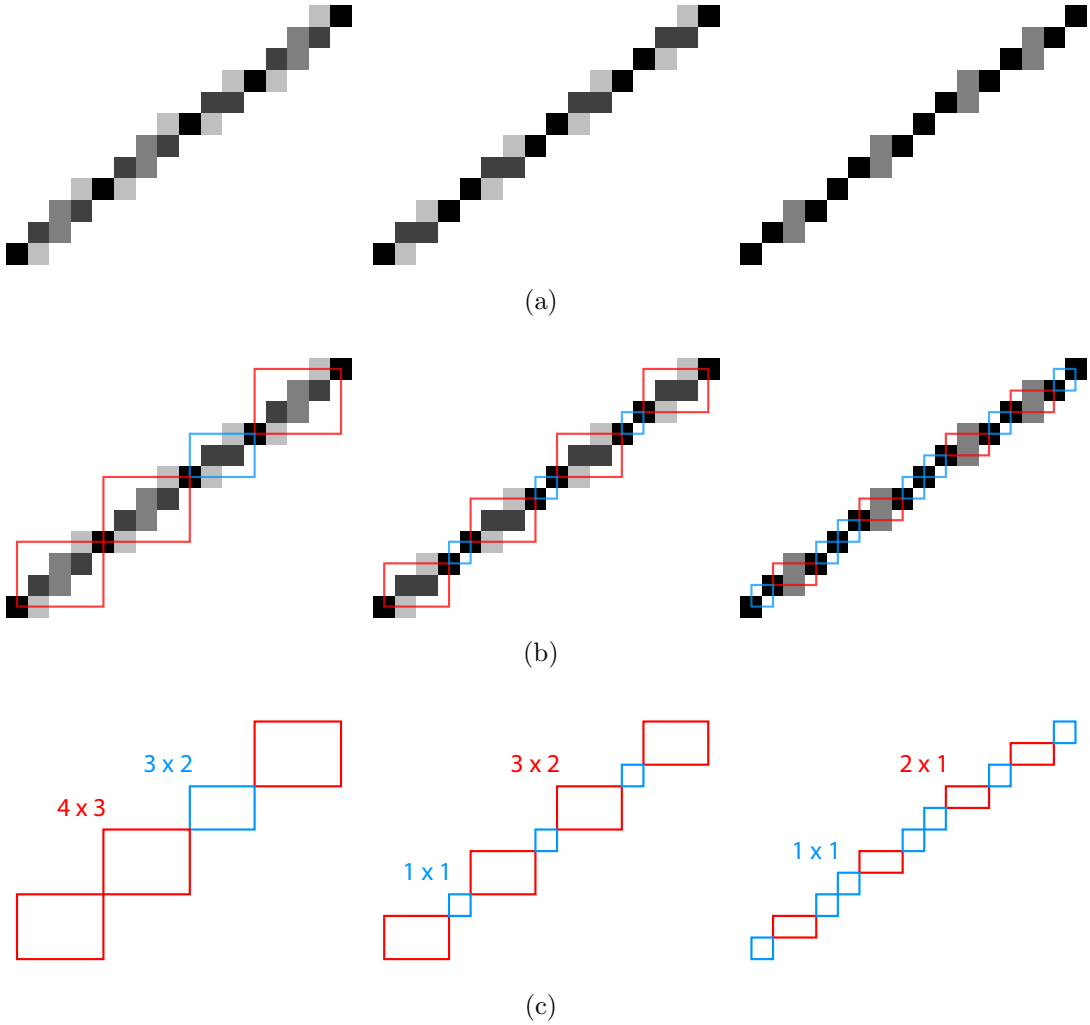
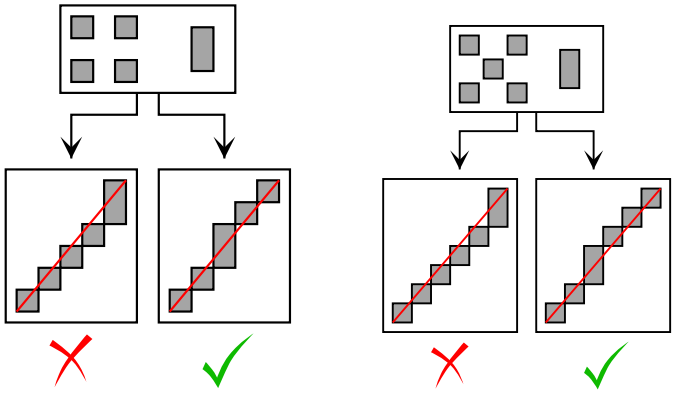


Figure 6.5: (a) From left to right, the amount of antialiasing applied to the line decreases. (b) The red and blue boxes indicate the two types of antialiased units used. (c) From left to right, the lengths of the antialiased units used also decrease.

summing their contributions in both the x- and y-directions, we get

$$A + B = 22, \tag{6.1}$$
$$A + 2B = 28. \tag{6.2}$$

Solving this set of equations gives us $A = 16$ and $B = 6$. Note that A and B always have integer solutions, given by Equations 6.13 and 6.14 in Section 6.2.3.



(a) (b)

Figure 6.6: (a) Given four length-1 units and one length-2 unit, the most symmetric arrangement is to put two length-1 units on either side of the length-2 unit. (b) Given five length-1 units and one length-2 unit, the most symmetric arrangement is to put two and three length-1 units on each side of the length-2 unit.

Next we wish to arrange these units to create a straight line. The idea is to arrange the two types of units such that the resulting line segment looks straight and symmetric. For example, Figure 6.6a shows that, given four length-1 units and one length-2 unit, the most symmetric way to arrange them is to put two length-1 units on either side of the length-2 unit. If we have five length-1 units instead (see Figure 6.6b), then the most symmetric arrangement puts two length-1 units on one side of the length-2 unit and three on the other side.

Returning to the example of a 22×28 line, we already know that 16 length-1 units and six length-2 units are needed. To arrange them as a pixelated line, we merge the units in a recursive fashion, creating several intermediate pixelated line segments, with the final segment being the line we want to draw.

The steps in this recursive algorithm are illustrated in Figure 6.7. In Step 0, we have 16 length-1 units and four length-2 units. In Step 1, we pair two length-1 units to each length-2 unit to create six pixelated line segments, with four length-1 units remaining. In Step 2, four of these pixelated segments are appended to four length-1 units to form four longer segments, with two of the shorter segments leftover. In Step 3, the four longer segments and two shorter segments are paired two-to-one to form two even longer segments. Finally, in Step 4, the two segments are joined to form the desired pixelated line consisting of 16 length-1 units and six length-2 units.

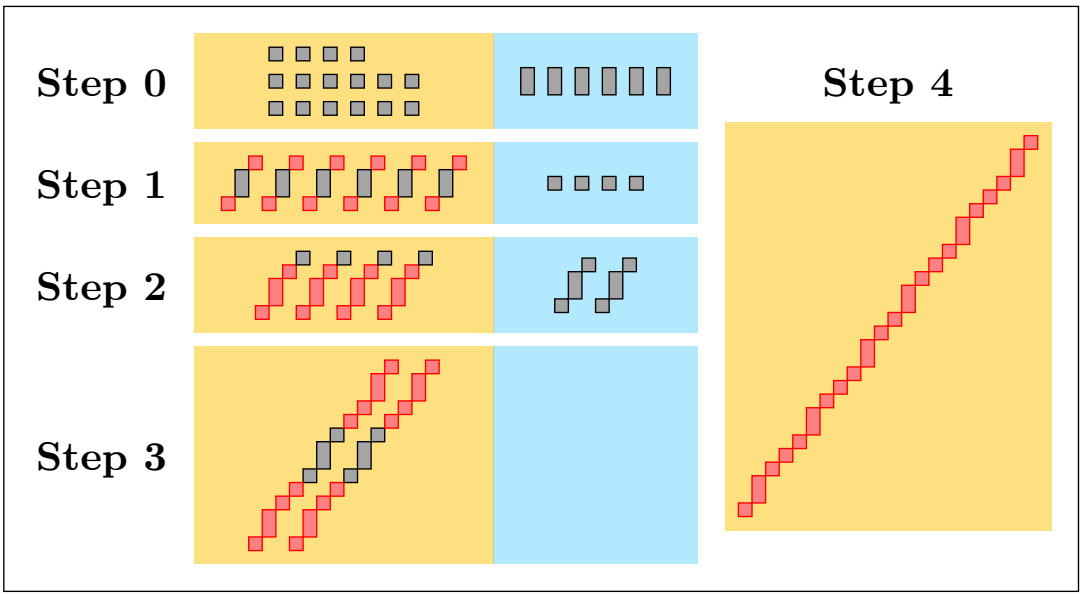


Figure 6.7: Steps in ELDA-A involve recursively building longer and longer line segments.

Before describing the details of the algorithm, we will define a numerical representation for pixelated lines and give a brief overview of the Euclidean algorithm, which is used as a subroutine for our line-drawing algorithm.

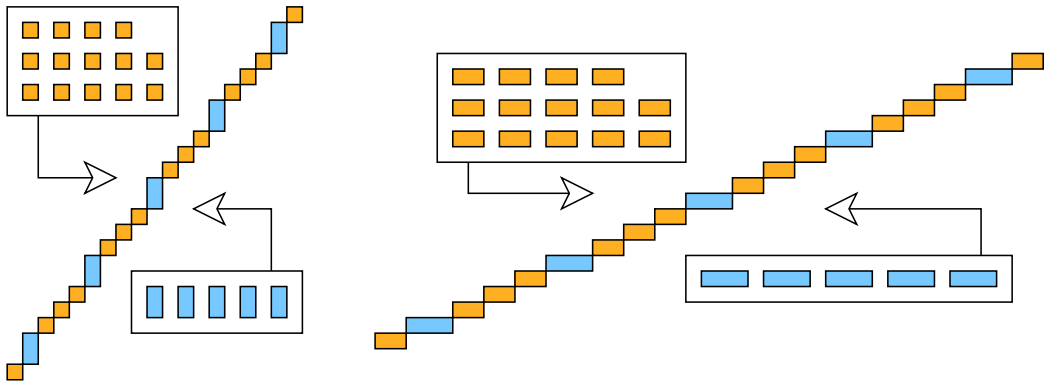
6.2.1 Binary Representation of a Pixelated Line

We will describe an aliased line using a *binary representation*. As mentioned earlier, ELDA-A constructs an aliased line with aliased units of length n and $n+1$. The formula for calculating n is provided in Section 6.2.3. We denote each of the shorter length-n units by 0 and each of the longer length-$(n+1)$ units by 1. Then the sequence of units in an aliased line can be rewritten as a binary sequence.

Figures 6.8a and 6.8b show two pixelated lines generated by the Euclidean Line-Drawing Algorithm from 14 shorter units and five longer units. Both lines have the binary representation

$$\{0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0\}. \tag{6.3}$$

because they are structurally identical. Even though the line in Figure 6.8b is longer, computing its unit structure takes the same amount of time as the shorter line in Figure 6.8a.



(a) (b)

Figure 6.8: Our algorithm ELDA-A does not differentiate between units of different sizes, only how many there are of each type. For example, (a) 14 length-1 units and five length-2 units are arranged in the same sequential order as (b) 14 length-2 units and five length-3 units.

Let us define three basic operations for pixelated lines: negation, addition, and scalar multiplication. Negating a line means reversing the order of the units. Adding two lines means appending the second line to the first. Multiplying a line by non-negative integer k means appending k copies of the line together, and multiplying a line by a negative integer k means multiplying it by $|k|$ and then negating the result. For example, if $\mathbf{c}_1 = \{0, 1, 0, 0\}$ and $\mathbf{c}_2 = \{1, 1, 0, 0\}$, then

$$-\mathbf{c}_1 = \{0, 0, 1, 0\}, \tag{6.4}$$

$$-\mathbf{c}_2 = \{0, 0, 1, 1\}, \tag{6.5}$$

$$\mathbf{c}_1 + \mathbf{c}_2 = \{0, 1, 0, 0, 1, 1, 0, 0\}, \tag{6.6}$$

$$3\mathbf{c}_1 = \{0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0\}, \tag{6.7}$$

$$-3\mathbf{c}_1 = \{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0\}. \tag{6.8}$$

Note that addition is not commutative in general. However, these operations can be combined and the addition of a negation can be abbreviated as subtraction. Another nice property is that multiplication and negation commute, so $-k\mathbf{c}$ is equivalent to $k(-\mathbf{c})$.

6.2.2 Euclidean Algorithm

In number theory, the Euclidean algorithm calculates the greatest common divisor (GCD) of two positive integers. Given two integers $a_0 \geq b_0 \geq 0$, we first compute their quotient q_0 and r_0 so that $a_0 = q_0 b_0 + r_0$. Then let $a_1 = b_0$ and $b_1 = r_0$, and repeat the process by finding their quotient and remainder. The algorithm yields the following sequence of equations:

$$\begin{aligned} a_0 &= q_0 b_1 + r_0, \quad (\text{let } a_1 = b_0, b_1 = r_0) && (6.9)\\ a_1 &= q_1 b_1 + r_1, \quad (\text{let } a_2 = b_1, b_2 = r_1) && (6.10)\\ a_2 &= q_2 b_2 + r_2, \quad (\text{let } a_3 = b_2, b_3 = r_2) && (6.11)\\ a_3 &= q_3 b_3 + r_3, \quad \ldots && (6.12) \end{aligned}$$

Eventually, a remainder r_N must equal zero, at which point the algorithm terminates and the final positive remainder r_{N-1} is the GCD of a_0 and b_0.

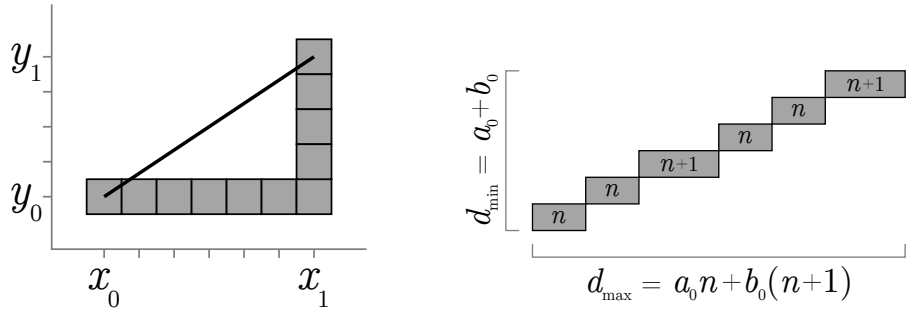
6.2.3 Algorithm Details

Now that we have defined the binary notation and some operations on it, we are ready to describe the details of ELDA-A. Suppose we wish to draw a line from pixel (x_0, y_0) to (x_1, y_1). Measuring corner-to-corner, the line has a width of $w = |x_1 - x_0| + 1$ and a height of $h = |y_1 - y_0| + 1$ (see Figure 6.9a). Now let $d_{\max} = \max(w, h)$ and $d_{\min} = \min(w, h)$. Then the aliased units used to draw the pixelated line will have lengths n and $n+1$, where $n = \left\lfloor \frac{d_{\max}}{d_{\min}} \right\rfloor$ (see Figure 6.9b). If there are a_0 shorter units and b_0 longer units, then we get the set of equations

$$\begin{aligned} n a_0 + (n+1) b_0 &= d_{\max} && (6.13)\\ a_0 + b_0 &= d_{\min}, && (6.14) \end{aligned}$$

with solutions given by $b_0 = d_{\max} - n d_{\min}$ and $a_0 = d_{\min} - b_0$. Note that b_0 is simply the remainder of $d_{\max}$ when divided by $d_{\min}$.

Next, we want to build up a series of intermediate segments from these units to produce the desired line. For simplicity, we will use $a_0 = 16$ and $b_0 = 6$ from the previous example. To start, we have a_0 longer units and b_0 shorter units. Denote the longer unit by $\mathbf{c}_{-2}$ and the shorter unit by $\mathbf{c}_{-1}$. The intermediate segments are denoted by $\mathbf{c}_0$, $\mathbf{c}_1$, etc. They are constructed by values computed in the Euclidean algorithm using a_0 and b_0 as inputs, as demonstrated in Table 6.1.



(a) (b)

Figure 6.9: (a) The initial set-up for ELDA-A. (b) How to calculate the number of each of the two units.

i	a_i	b_i	q_i	r_i	$\mathbf{c}_i$
-2					$\{1\}$
-1					$\{0\}$
0	16	6	2	4	$\{0,1,0\}$
1	6	4	1	2	$\{0,1,0,0\}$
2	4	2	2	0	$\{0,1,0,0,0,1,0,0,0,1,0\}$
3	2	0			$\{0,1,0,0,0,1,0,0,0,1,0,0,1,0,0,0,1,0,0,0,1,0\}$

Table 6.1: Variables in each step of the recursion. At Step i, $\{a_i, b_i, q_i, r_i\}$ are values computed by the Euclidean algorithm, and $\mathbf{c}_i$ is the binary representation of the intermediate segment created.

The table shows values (a_i, b_i, q_i and r_i) computed at each step of the Euclidean algorithm as well as the intermediate segment $\mathbf{c}_i$ constructed at that step using the following recurrence relation:

$$\mathbf{c_i} \quad = \quad \begin{cases} \left\lceil \frac{q_i}{2} \right\rceil \mathbf{c}_{i-1} + \mathbf{c}_{i-2} - \left\lfloor \frac{q_i}{2} \right\rfloor \mathbf{c}_{i-1} & \text{if } b_i \neq 0, \\ \left\lceil \frac{a_i}{2} \right\rceil \mathbf{c}_{i-1} - \left\lfloor \frac{a_i}{2} \right\rfloor \mathbf{c}_{i-1} & \text{if } b_i = 0. \end{cases} \tag{6.15}$$

Remember that this formula uses the string manipulation functions defined in Section 6.2.1, and not normal addition and multiplication. The recurrence relation says that when $b_i \neq 0$, we have q_i copies of $\mathbf{c}_{i-1}$ and one of $\mathbf{c}_{i-2}$; we take $\left\lceil \frac{q_i}{2} \right\rceil$ copies of $\mathbf{c_i}$, append $\mathbf{c}_{i-2}$ to it, followed by $\left\lfloor \frac{q_i}{2} \right\rfloor$ negated copies of $\mathbf{c_i}$, as illustrated in Figure 6.10a. As for when $b_i = 0$, this is the terminating condition when we only have a_i copies of $\mathbf{c}_{i-1}$. In this case, append $\left\lfloor \frac{a_i}{2} \right\rfloor$ negated copies to $\left\lceil \frac{a_i}{2} \right\rceil$ regular copies to get $\mathbf{c_i}$.

In the recurrence relation, we use negation to maximize the symmetry in the resulting pixelated line. Dividing the segments into two groups and appending one group at the start with the other group at the end also preserves symmetry. The choice of where we put the floor and ceiling functions is arbitrary. Swapping them would work equally well in preserving symmetry.

One problem with the current definition is that performing the negation at every step is costly. Instead, we will calculate the negated segments recursively as well. Let ℓ be the total number of steps in the algorithm (for example, $\ell = 4$ for the example shown in Figure 6.7), then we have the following recurrence relations:

$$\mathbf{c}_i = \begin{cases} \{1\} & \text{if } i = -2, \\ \{0\} & \text{if } i = -1, \\ \left\lceil \frac{q_i}{2} \right\rceil \mathbf{c}_{i-1} + \mathbf{c}_{i-2} + \left\lfloor \frac{q_i}{2} \right\rfloor (-\mathbf{c}_{i-1}) & \text{if } 0 \geq i < \ell, \\ \left\lceil \frac{a_i}{2} \right\rceil \mathbf{c}_{i-1} + \left\lfloor \frac{a_i}{2} \right\rfloor (-\mathbf{c}_{i-1}) & \text{if } i = \ell, \end{cases} \tag{6.16}$$

and

$$-\mathbf{c}_i = \begin{cases} \{1\} & \text{if } i = -2, \\ \{0\} & \text{if } i = -1, \\ \left\lfloor \frac{q_i}{2} \right\rfloor \mathbf{c}_{i-1} + (-\mathbf{c}_{i-2}) + \left\lceil \frac{q_i}{2} \right\rceil (-\mathbf{c}_{i-1}) & \text{if } 0 \geq i < \ell, \\ \left\lfloor \frac{a_i}{2} \right\rfloor \mathbf{c}_{i-1} + \left\lceil \frac{a_i}{2} \right\rceil (-\mathbf{c}_{i-1}) & \text{if } i = \ell. \end{cases} \tag{6.17}$$

The recursion is illustrated in Figures 6.10b and 6.10c. Figure 6.11 shows how successive values of $\mathbf{c}_i$ are constructed recursively.

6.2.4 Error Bound

Our algorithm creates lines with highly regular pixel patterns, which is good for pixel art purposes, but for it to be a good general purpose rasterization algorithm, it must also produce reasonable approximations. In Appendix E, we present a lengthy derivation of an error bound. Here we will provide a number theoretic argument for why ELDA-A produces good pixelated approximations for any line.

We know that at Step i, the algorithm constructs an intermediate segment $\mathbf{c}_i$. Let m_i and n_i denote, respectively, the number of shorter and longer units in $\mathbf{c}_i$. Table 6.2 augments Table 6.1 with additional columns showing values of m_i and n_i. In each row, m_i is the number of 0s in $\mathbf{c}_i$ and n_i is the number of 1s in $\mathbf{c}_i$. Even though $\{m_i\}$ and $\{n_i\}$

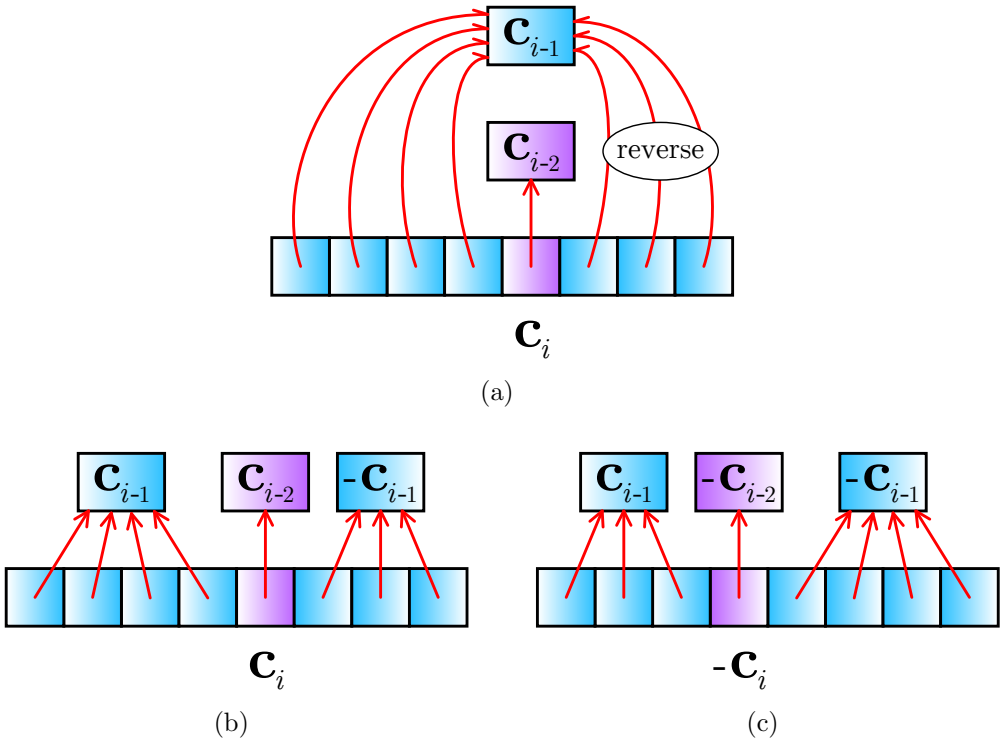


Figure 6.10: (a) At Step i of the recursion, $\mathbf{c}_i$ consists of one $\mathbf{c}_{i-2}$ and several $\mathbf{c}_{i-1}$, some of which are reversed. (b, c) To reduce the number of calculations, keep track of the reversed segment at each step. The illustrations show how to construct $\mathbf{c}_i$ and $-\mathbf{c}_i$.

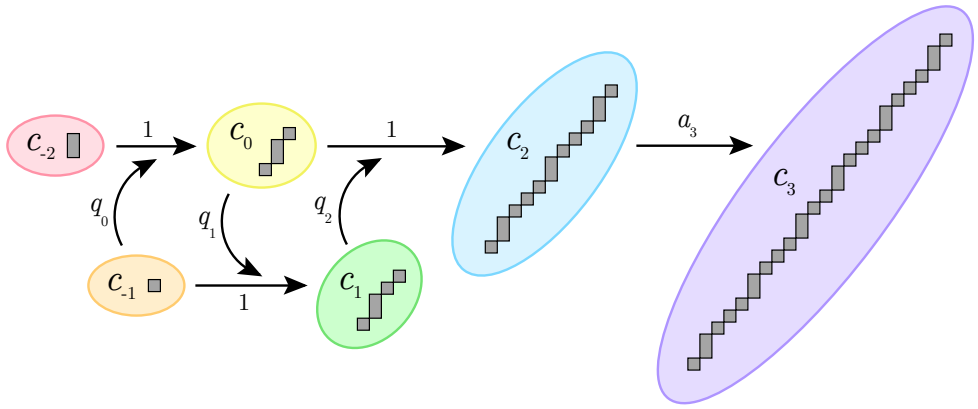


Figure 6.11: The recursive construction of intermediate segments in ELDA-A.

have different initial values, they satisfy the same recurrence relations $m_i = m_{i-2} + q_i m_{i-1}$ and $n_i = n_{i-2} + q_i n_{i-1}$.

It turns out that, out of all fractions with denominator up to n_i, $\frac{m_i}{n_i}$ is the closest to $\frac{a_0}{b_0}$. The reason is that, since our algorithm relies on the Euclidean algorithm, we can write $\frac{a_0}{b_0}$ a continued fraction:

$$\frac{a_0}{b_0} = q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{\ddots + \cfrac{1}{q_\ell}}}}, \tag{6.18}$$

and m_i and n_i are called *convergents* that satisfy

$$\frac{m_i}{n_i} = q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{\ddots + \cfrac{1}{q_i}}}}. \tag{6.19}$$

A well-known theorem [12] in number theory says that $\frac{m_i}{n_i}$ is the best rational approximation of $\frac{a_0}{b_0}$ with denominator up to n_i. In addition,

$$\left| \frac{a_0}{b_0} - \frac{m_i}{n_i} \right| < \frac{1}{n_i^2}. \tag{6.20}$$

Therefore, all the intermediate line segments constructed in our algorithm are good approximations for the slope of the final pixelated line, and the approximation improves with each step.

6.3 Euclidean Line-Drawing Algorithm - Antialiased

Now we describe the antialiased version of our algorithm, ELDA-AA. The purpose of ELDA-AA is to mimic manual antialiasing of straight lines. Similarly to ELDA-A, each antialiased line constructed by ELDA-AA is also composed of two different types of units and their arrangement is determined by the Euclidean algorithm. However, ELDA-A and ELDA-AA have several differences:

i	a_i	b_i	q_i	m_i	n_i	$\mathbf{c}_i$
-2				0	1	$\{1\}$
-1				1	0	$\{0\}$
0	16	6	2	2	1	$\{0,1,0\}$
1	6	4	1	3	1	$\{0,1,0,0\}$
2	4	2	2	8	3	$\{0,1,0,0,0,1,0,0,0,1,0\}$
3	2	0		16	6	$\{0,1,0,0,0,1,0,0,0,1,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0\}$

Table 6.2: Variables in each step of the recursion in ELDA-A. At Step i, $\{a_i, b_i, q_i, r_i\}$ are values computed by the Euclidean algorithm. m_i and n_i are the i-th convergents, representing respectively the number of 0s and 1s in the binary representation of $\mathbf{c}_i$, the intermediate segment constructed at Step i.

1. Drawing an aliased unit is trivial since it is simply a pixel span. In comparison, drawing an antialiased unit is nontrivial since it requires determining the opacities of many pixels.

2. In the aliased case, the width and height of the line determine exactly which two units to use, but in the antialiased case, there may be more than one possibility for the pair of units to use.

3. ELDA-AA has several tunable parameters to allow for different styles of manual antialiasing.

To address these differences, we describe how to determine the pixel opacities in antialiased units in Section 6.3.1, how the unit sizes affect the resulting line in Section 6.3.2, and what parameters the algorithm uses in Section 6.3.3.

6.3.1 Drawing Antialiased Units by Wu's Algorithm

We will start by describing how to draw individual antialiased units with a line thickness of 1 using Wu's antialiasing algorithm for lines [60]. Suppose we want to draw a unit with width w and height h, where $w \leq h$ (see Figure 6.12a). This line intersects several horizontal pixel pairs (see Figure 6.12b). We compute the horizontal distance from each pixel centre to the line (see Figure 6.12c). For each pixel pair, suppose the distances from the pixel centres to the line are d_1 and d_2. Then the opacity of the pixels are set to $1 - d_1$ and $1 - d_2$, respectively. In other words, the closer a pixel is to the line, the higher its

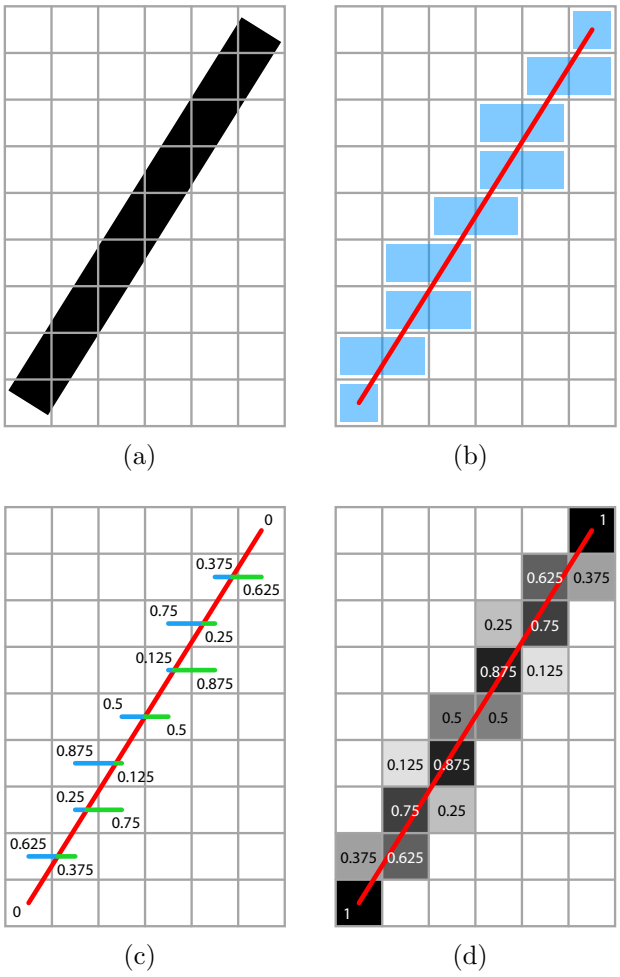


Figure 6.12: Steps in Wu's antialiasing algorithm: (a) the input line has a thickness of 1, (b) the line (in red) intersects several horizontal pixel pairs (in blue), (c) the horizontal distance from each pixel to the line is calculated, and (d) the opacity of a pixel is computed as one minus this distance.

opacity. The endpoints are set to 100% opacity since they are exactly on the line. Since the antialiased line produced is rotationally symmetric, we only need to compute the opacities for half of the pixels. If the line we want to draw is more wide than tall (i.e., $w > h$), then we consider vertical pixel pairs instead and calculate their vertical distances to the line.

6.3.2 Determining the Sizes of Antialiased Units

When drawing a line with ELDA-A, we use its width and height to determine the lengths of the two aliased units to use. With ELDA-AA, the lengths of the two antialiased units are not uniquely defined by the width and height of the line. We will discuss how to select the appropriate antialiased units to use in this section.

First let us standardize some terminology for discussing antialiased units. Up to now, we have been referring to aliased units by their lengths because they are always either one pixel wide or one pixel tall. Antialiased units, on the other hand, do not have such restrictions. We refer to the *size* of an antialiased unit as $a \times b$ where a is its width and b is its height (measured centre-to-centre). Its *length* is then defined as $\max(a, b)$.

Now let us describe the algorithm with an example. Suppose we wish to draw a line with a width of 22 and a height of 28. In ELDA-A, this line would be composed of 16 length-1 vertical units and six length-2 vertical aliased units. The sizes of these units are 1×1 and 1×2. For now, let these be the sizes of the antialiased units. These two units are shown in Figure 6.13 as $\mathbf{c}_{-2}$ and $\mathbf{c}_{-1}$, drawn using Wu's algorithm. Notice that since antialiased units measured centre-to-centre, they look slightly larger than aliased units of the same sizes. Now given these two units, we can join them together in the same recursive way as in ELDA-A, as illustrated in Figure 6.13.

The resulting line contains little antialiasing. The amount of antialiasing is measured in terms of how many pixels are used, and in this case, the antialiased line contains only a few more pixels than the aliased line (see Figure 6.11). Using small amounts of antialiasing may be good in some cases when the palette size is severely limited, but the less antialiasing used, the more jagged the line appears. We want to control exactly how much antialiasing to use by being able to increase the amount of antialiasing in small increments if desired.

One solution is to use larger antialiased units to build the pixelated line. The sizes of the units should not be chosen arbitrarily, however, because they may not tile the line exactly. The recursive nature of the algorithm provides a convenient way of choosing the unit sizes since the sizes of any two consecutive intermediate segments would tile the line exactly. For example, we could use $\mathbf{c}_0$ and $\mathbf{c}_1$ to do the construction (see Figure 6.14), and the resulting line would contain more antialiasing. In general, using larger antialiased units results in lower error, and the line appears smoother due to the increased antialiasing. However, using smaller antialiased units promotes higher regularity in the pixel pattern and also uses fewer colours.

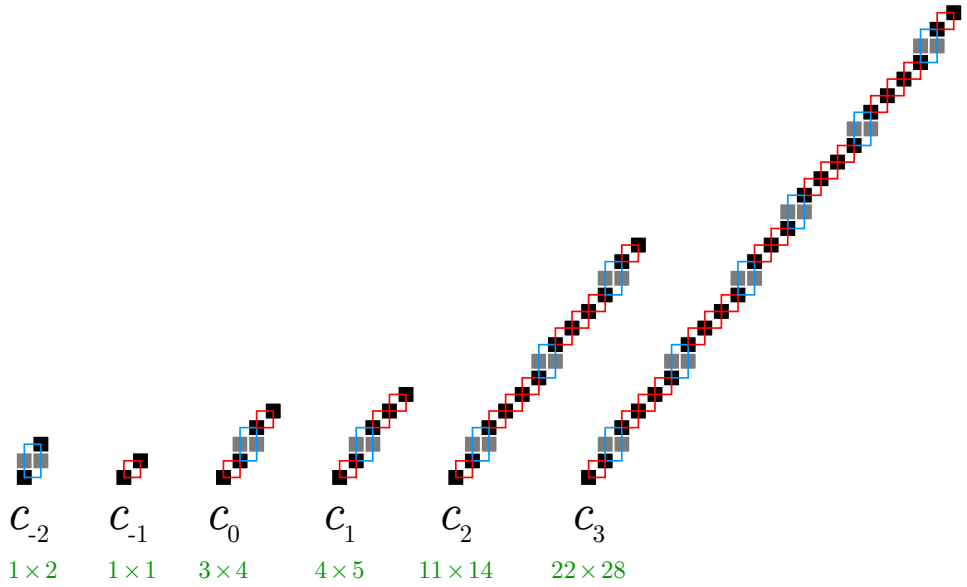


Figure 6.13: Building a minimally antialiased line from 1×1 and 1×2 units.

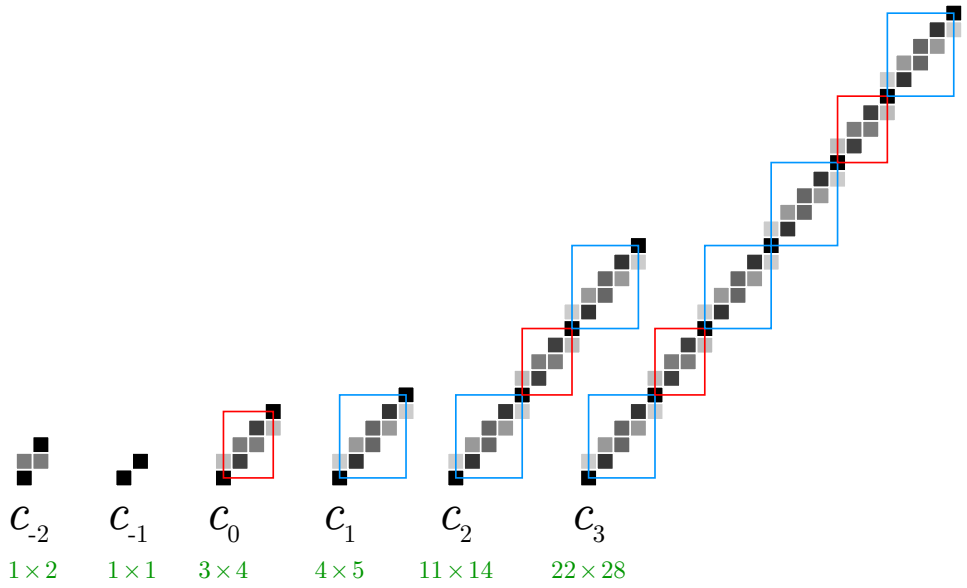


Figure 6.14: Building a moderately antialiased line from 3×4 and 4×5 units.

6.3.3 Antialiasing Parameters

When learning about manual antialiasing from various pixel art tutorials, we found many variations such as the number of colours used, the apparent thickness of the lines drawn, and the amount of antialiasing applied. This observation suggests that there are many degrees of freedom in antialiasing, and we would like to add several parameters to ELDA-AA to reflect these options.

We will introduce five parameters: (1) error threshold, (2) maximum unit length, (3) apparent thickness, (4) radius, and (5) palette size. We will first define a set of default parameter values, and then show the effect of modifying each parameter individually. Finally we will show how the parameters can work together to produce certain effects.

One parameter we can use is the error threshold. The shorter the units are, the worse they approximate the line we want to draw. By putting a threshold on the error, we ensure the quality of the approximation. Between a unit and the line to be drawn, we calculate the error as the angle difference between them. Then we find the shortest pair of antialiased units such that both have error values below a chosen threshold. The default value for the error threshold is 0, and the first row in Figure 6.15 shows how increasing the parameter value changes an antialiased line.

Another useful parameter is the maximum unit length. It places a threshold on the lengths (the width or height, whichever is longer) of the antialiased units used. As discussed in Section 6.3.2, increasing the threshold encourages the use of longer units, which provides better approximations by applying more antialiasing, but at the cost of lower regularity in pixel pattern and more colours used. On the other hand, lowering the threshold shortens the units, thereby increasing the regularity in pixel pattern and reducing the number of colours, at the cost of the line looking more jagged due to less antialiasing. There is an aesthetic trade-off in the choice of threshold, and the best parameter value depends on the context in which it is used. The default value for maximum unit length is ∞, and the second row in Figure 6.15 shows how decreasing the parameter value changes an antialiased line. For each line, the red and blue boxes indicate the types of units used.

Two more parameters deal with stroke width: apparent thickness and radius. Apparent thickness, as defined previously, involves manipulating pixel opacity to change how thick a line appears. This parameter scales the opacity values of all the pixels by a factor of r where $0 \leq r \leq 1$ so that the resulting line appears to be r units thick. The default value for apparent thickness is 1, and the third row in Figure 6.15 shows how decreasing the parameter value changes an antialiased line.

The radius parameter controls the actual pixel coverage. The greater the radius, the more blurry a line looks. The default value of the radius is 1 because currently, using Wu's algorithm, all the pixels are at most a distance of 1 from the line (the distances are measured as shown in Figure 6.12b). By decreasing the radius, some of the lower opacity pixels will not be drawn. The radius ranges from 0.5 to 1, because if it is less than 0.5, then the line starts becoming disconnected with dropouts. The default value for the radius is 1, and the fourth row in Figure 6.15 shows how decreasing the parameter value changes an antialiased line.

Finally, we have palette size, a parameter that is crucial to pixel art due to its palette restriction. Currently, we use linear greyscale values. That is, if the palette size is n, then the palette consists of blacks with opacity values $\frac{0}{n-1}$, $\frac{1}{n-1}$, $\ldots$, $\frac{n-2}{n-1}$, $\frac{n-1}{n-1}$. The palette reduction is applied by mapping each pixel's colour to the closest greyscale value in the given palette. The default value for palette size is ∞, and the fifth row in Figure 6.15 shows how decreasing the parameter value changes an antialiased line.

Now let us look at the effect of all five parameters together in Figure 6.15. First, notice that decreasing the maximum unit length has the same effect as increasing the error threshold; this is reasonable because longer units have lower error values. Since the two parameters control the same thing, we will keep the maximum unit length parameter and discard the error threshold parameter since the former is easier to compute and more intuitive compared to the latter.

Changing the palette size is interesting because the colour reduction causes pixel patterns to emerge, as indicated by coloured boxes. These patterns look slightly different from the antialiased units used before. Unlike the antialiased units, these pixel patterns are not precomputed. Therefore, even though decreasing the palette size produces interesting pixel patterns, it does not improve the performance of the algorithm.

Varying the apparent thickness and radius work intuitively, but when the palette size is unrestricted, decreasing the apparent thickness does nothing more than lighten the pixels. It is more interesting when the palette size is restricted. Figure 6.16 shows the effect of varying the apparent thickness when the palette size is restricted to at most five colours. Now interesting pixel patterns emerge at a result of trying to lighten the pixels while staying within the 5-colour palette restriction.

6.3.4 Antialiasing with a Custom Palette

So far we have used greyscale palettes for antialiasing. The reason for using a limited set of colours in manual antialiasing is to avoid introducing too many unnecessary colours to

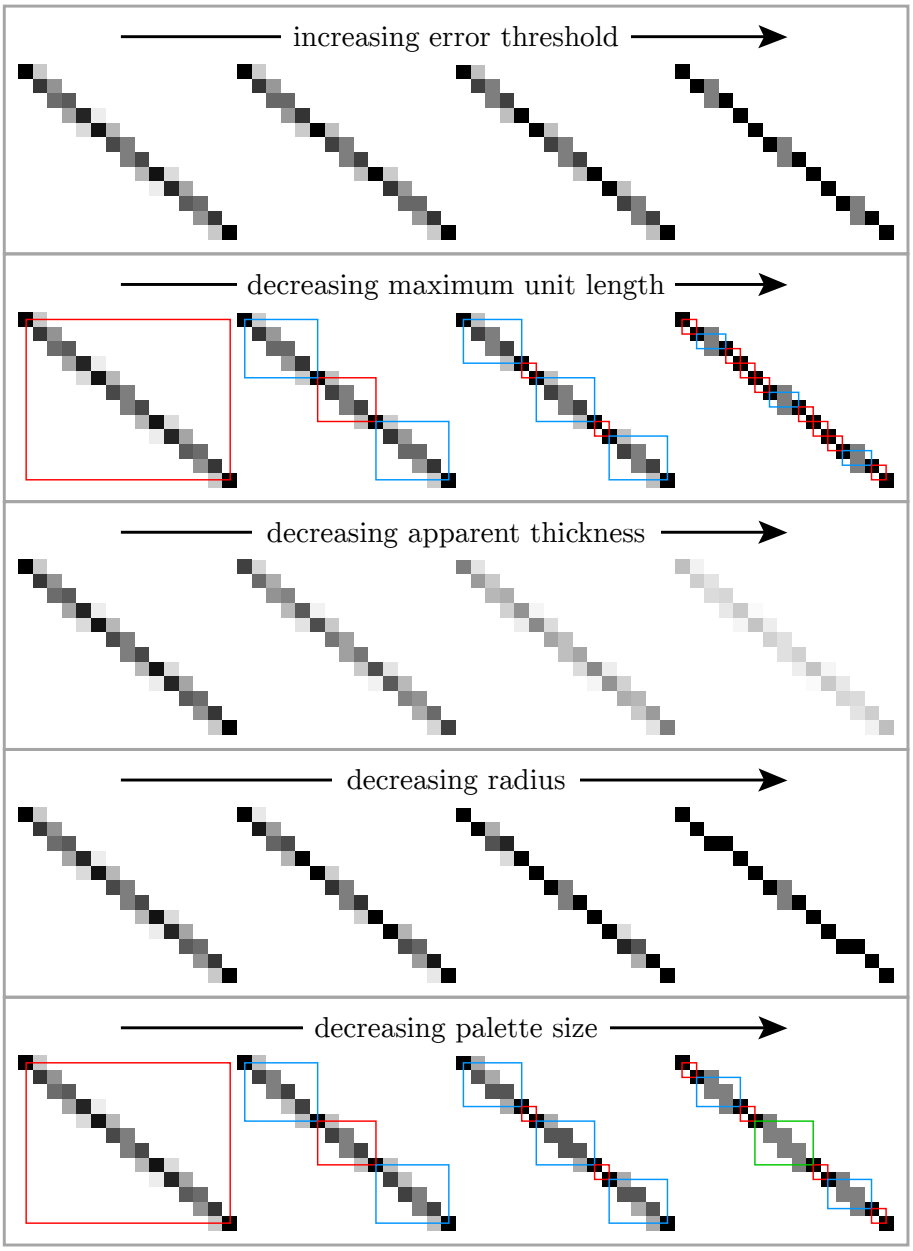


Figure 6.15: How the antialiasing changes as one parameter changes while others set to their default values.

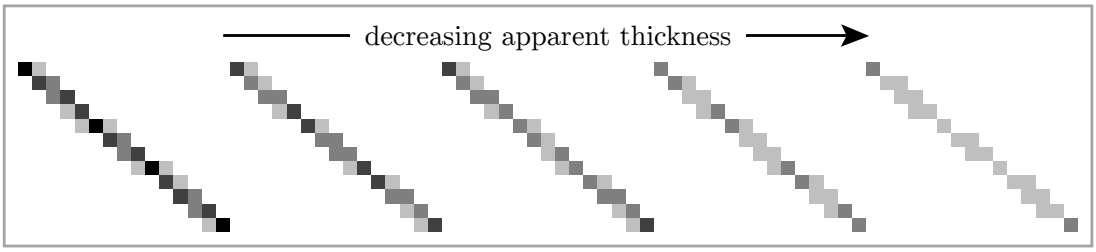


Figure 6.16: Varying the apparent thickness when restricted to a 5-colour palette. The other parameters are set at their default values.

the existing palette. Since the existing palette usually contains non-greyscale colours, we want to extend ELDA-AA to be able to use non-greyscale colours for antialiasing.

In Ruthner's antialiasing tutorial, he explains that a non-greyscale palette can be treated as a greyscale palette by considering the "value" of each colour. In this case, "value" does not necessarily refer to the value of a colour from the HSV colour model. Instead, it is problem of human perception: given a non-greyscale colour, what greyscale value appears closest to it?

There are many ways to convert colours to grey levels. We experimented with five different measures from various colour models: intensity I from the HSI model [24], value V from the HSV model [51], lightness L from the HSL model [38], and luma Y' from the Y′UV model. Luma is calculated as the weighted average of gamma-corrected RGB values based on perceived luminance. The Rec. 601 and Rec. 709 standards—commonly used for standard-definition and high-definition television formats respectively [54, 55]—use different weights to calculate the Rec. 601 luma Y'_{601} and the Rec. 709 luma Y'_{709}. Here are all five greyscale values calculated from RGB values that range from 0 to 1:

$$I = \frac{R+G+B}{3}, \tag{6.21}$$

$$V = \max(R,G,B), \tag{6.22}$$

$$L = \frac{\max(R,G,B)+\min(R,G,B)}{2}, \tag{6.23}$$

$$Y'_{601} = 0.299R + 0.587G + 0.114B, \tag{6.24}$$

$$Y'_{709} = 0.2126R + 0.7152G + 0.0722B. \tag{6.25}$$

We used these five methods to convert a non-greyscale palette to various greyscale palettes, and then apply ELDA-AA as before.

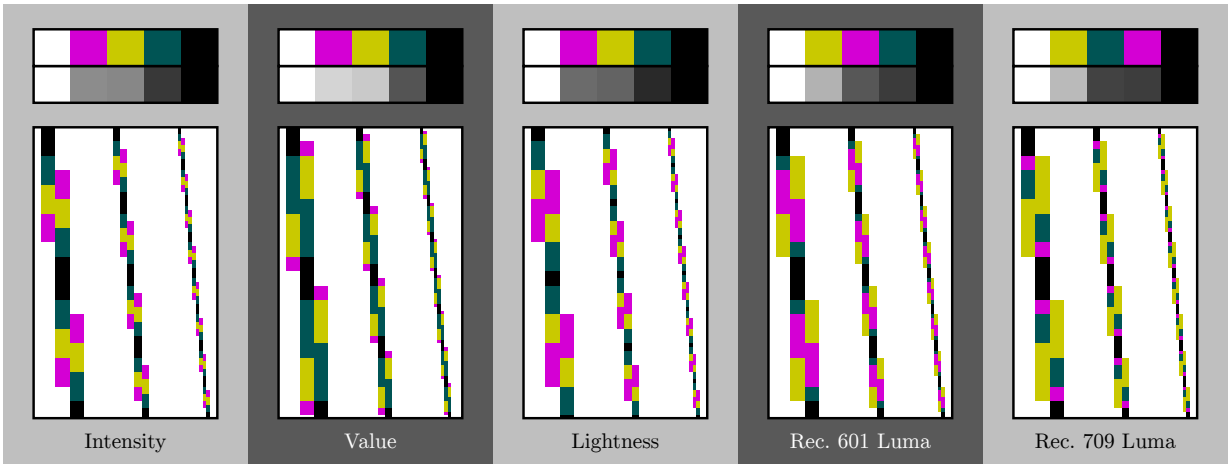


Figure 6.17: Five different greyscale conversions are applied to a non-greyscale palette then used for manual antialiasing. For each greyscale conversion, both the coloured and greyscale palette are arranged in decreasing intensity according to the greyscale palette produced.

The results are shown in Figure 6.17 at three different resolutions. Above each set of lines, we have the non-greyscale palette and its corresponding greyscale palette, both arranged from lightest to darkest according to the greyscale values. On our monitor, the colours from lightest to darkest are in this order: white, yellow, magenta, green, and black. This ordering is consistent with the ordering computed using Rec. 601 luma. As a result, the antialiased lines it draws appear to be the smoothest. In contrast, the result obtained by using value, for example, contains magenta pixels that stand out distinctly from the line instead of merging with the other pixels to create smooth antialiasing. Our experiment shows that different greyscale conversions can produce antialiasing of different quality. On our monitor, Rec. 601 luma is the best greyscale conversion but it may be different on another display and possibly for other viewers.

Using Rec. 601 luma as the greyscale conversion scheme, we compare our results to those created by Ruthner in his antialiasing tutorial in Figure 6.18. There are three lines of slope $\frac{3}{2}$, each drawn with a 5-colour palette, shown at three different resolutions for comparison. ELDA-AA and the pixel artist use distinctly different pixel patterns to construct the lines, but in both cases, the repeating units are 2×3, demonstrating our success in achieving a line drawn with highly regular pixel patterns. As for colour selection, both ELDA-AA and Ruthner used only four out of five colours to draw each line. Their choice of colours differ for the green line: ELDA-AA chose not to use the light green while Ruthner chose

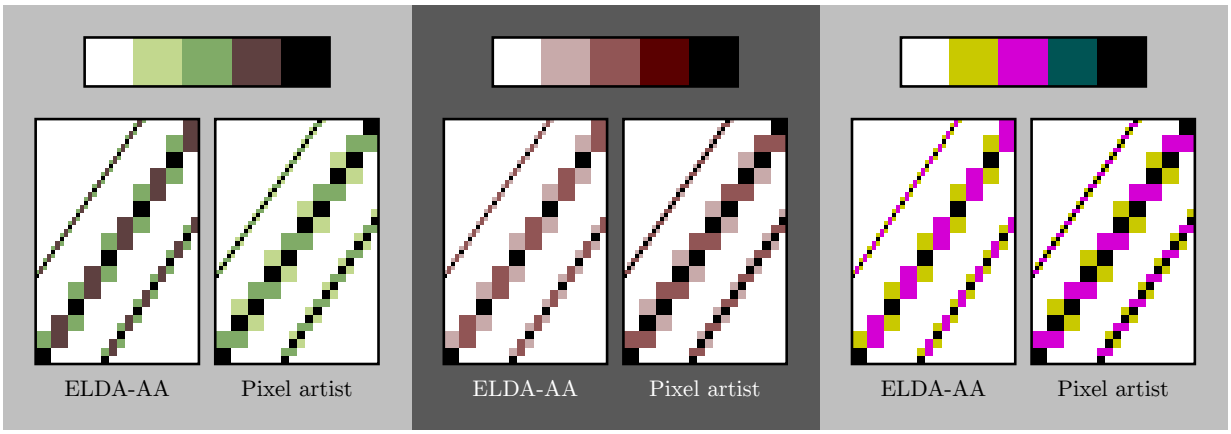


Figure 6.18: A comparison of how ELDA-AA and a pixel artist applies manual antialiasing to a line given three different non-greyscale palettes.

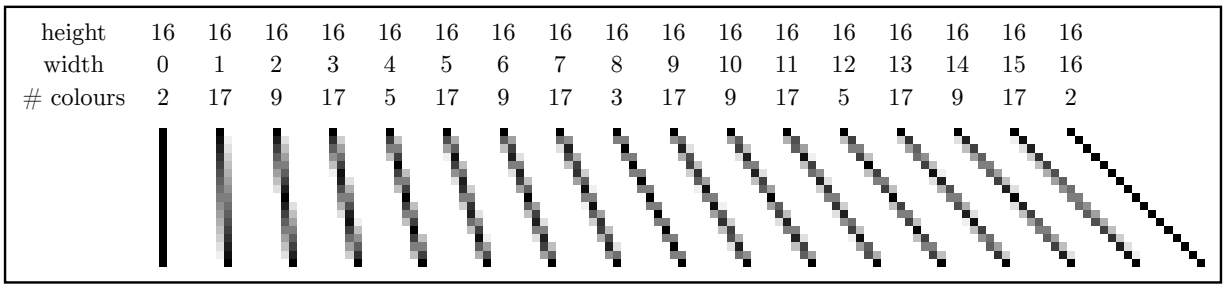


Figure 6.19: 17 antialiased lines with a height of 16 pixels and widths varying from 0 to 16 pixels. The number of colours used to antialias these lines varies significantly.

not to use the brown. The third difference is pixel coverage. ELDA-AA covers fewer pixels than the pixel artist does overall, but we believe our results are sufficiently antialiased to remove jaggedness.

6.4 Quality of Pixelated Lines

Our line-drawing algorithms provide a high level of regularity in pixel patterns compared to other line-drawing algorithms. However, the quality of a line is highly dependent on its slope. For example, a vertical line can be drawn with minimal antialiasing whereas a near-vertical line requires a lot of antialiasing.

In this section, we present a way of measuring quality for both aliased and antialiased lines. Roughly speaking, the quality of a line refers to its regularity in pixel patterns. Since our algorithm deals explicitly with pixel pattern regularity, it provides an easy way to measure quality. Being able to calculate the quality quickly allows us to snap to high-quality lines at interactive speed.

First we will describe what quality means for antialiased lines. Figure 6.19 shows 17 lines with height 16 and widths ranging from 0 to 16, drawn using ELDA-AA with default parameters (which is equivalent to Wu's algorithm). The result is a set of antialiased lines drawn using various shades of grey. Since there are no artificial constraints on the number of colours, the lines drawn with fewer colours are inherently of higher quality because palette reduction can be applied with less sacrifice to the accuracy of the approximation.

We consider the quality of an antialiased line to be inversely proportional to the number of colours it uses when antialiased with ELDA-AA with neither palette size constraints nor maximum unit length constraints (in other words, when ELDA-AA is equivalent to Wu's algorithm). By simplifying our algorithm, we discover that

$$n_c = \frac{\max(|w|, |h|)}{\gcd(|w|, |h|)} + 1, \tag{6.26}$$

where n_c is the number of colours used, and w and h are the width and height of the line.

Although n_c is related to the quality of a line, we prefer a measure of quality that lies between 0 and 1, with which a higher value indicates a higher line quality. Therefore we propose the following measure of quality for an antialiased line:

$$Q_{\mathrm{AA}}(w, h) = \frac{\gcd(|w|, |h|)}{\max(|w|, |h|)}. \tag{6.27}$$

This value is inversely proportional to the number of colours used, and covers the interval $(0, 1]$.

Figure 6.20 shows a heat map of $Q_{\mathrm{AA}}(w, h)$ centred at $(w, h) = (0, 0)$, in which the intensity at (w, h) corresponds to the value of $Q_{\mathrm{AA}}(w, h)$. The white lines (i.e., vertical, horizontal and 45° diagonal lines) are those can be drawn at the highest quality. Lines with slope n and $1/n$ for some integer $n \geq 2$ (i.e., perfect lines), are also relatively bright, but as the value of n increases, the brightness decreases because it takes more colours to antialias them, which lowers the quality. Other bright lines include those with "good" slopes such as 2/3 and 3/4. Although these lines are not perfect, they can be drawn with short antialiased units and therefore use fewer colours.

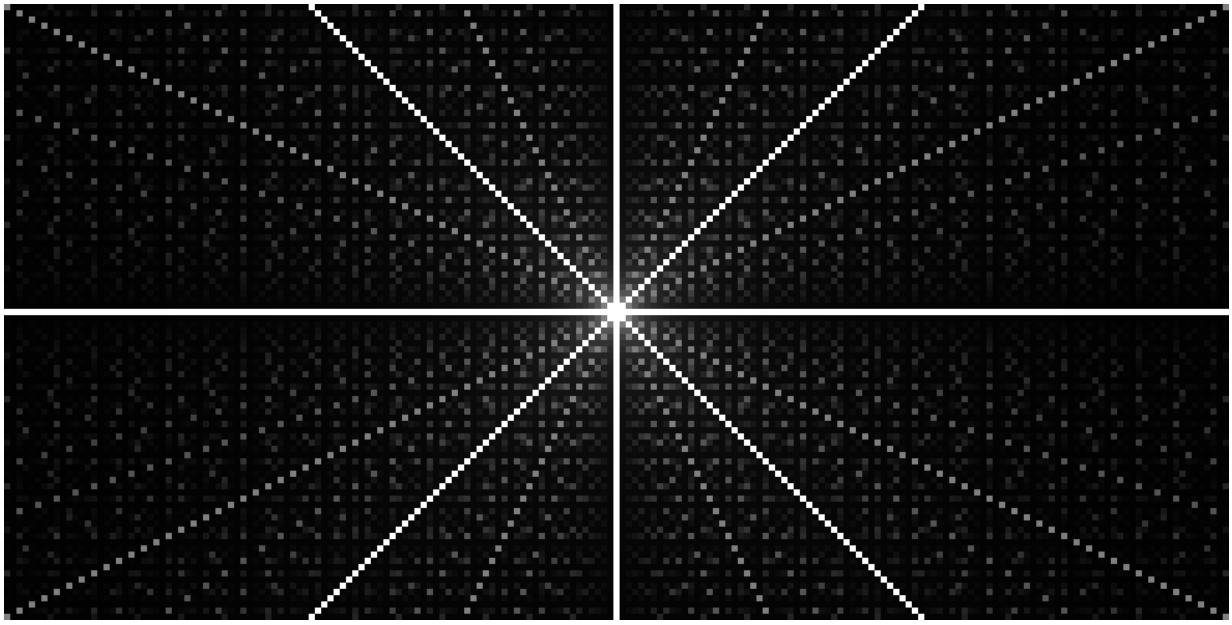


Figure 6.20: A heat map of the quality of antialiased lines using ELDA-AA. The centre of the image is $(0, 0)$. The intensity of pixel (w, h) represents the quality $Q_{\mathrm{AA}}(w, h)$ of a line with width w and height h. The intensity values are normalized to the range $[0, 1]$.

This quality measure applies similarly to aliased lines, but with an offset of 1 in both w and h since aliased lines are measured centre-to-centre rather than corner-to-corner. The quality Q_{A} of an aliased line is given by

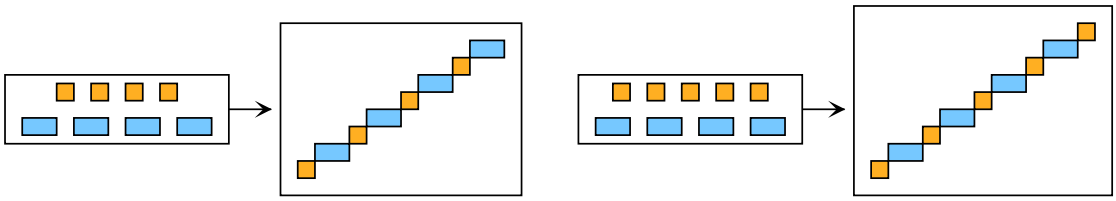
$$Q_{\mathrm{A}}(w, h) = \frac{\gcd(|w| + 1, |h| + 1)}{\max(|w| + 1, |h| + 1)}. \tag{6.28}$$

Since Q_{A} is a relative measure of quality rather than an absolute one, we can remove the 1 in the denominator to get

$$Q_{\mathrm{A}}(w, h) = \frac{\gcd(|w| + 1, |h| + 1)}{\max(|w|, |h|)}. \tag{6.29}$$

This expression can be simplified to

$$Q_{\mathrm{A}}(w, h) = \frac{\gcd(A, B)}{\max(|w|, |h|)}, \tag{6.30}$$



(a) (b)

Figure 6.21: (a) An aliased line composed of four length-1 units and four length-2 units and (b) an aliased line composed of five length-1 units and four length-2 units appear visually similar.

where A and B are the number of shorter and longer aliased units in the line. The derivation is as follows (note that n and $n+1$ are the lengths of the two units):

$$\begin{aligned}
\gcd(|w|+1, |h|+1) &= \gcd(A+B, nA+(n+1)B) && (6.31)\\
&= \gcd(A+B, nA+(n+1)B-n(A+B)) && (6.32)\\
&= \gcd(A+B, B) && (6.33)\\
&= \gcd((A+B)-B, B) && (6.34)\\
&= \gcd(A, B). && (6.35)
\end{aligned}$$

There is a significant difference between the quality of antialiased and aliased lines. Suppose we have a line composed of $A = 4$ length-1 units and $B = 4$ length-2 units, as shown in Figure 6.21a. Because $A = B$, Q_{A} is high since the GCD of A and B is maximized. On the other hand, if $A = B+1 = 5$, then we would have the line as shown in Figure 6.21b. Although this line appears visually similar to the previous line, its quality Q_{A} is low since $\gcd(A, B) = 1$. We would like the two lines to have the same quality since they are both drawn by alternating the two types of units.

The following revised quality measure successfully corrects for this discrepancy:

$$Q_{\mathrm{A}}(w, h) = \frac{\max(g_1, g_2, g_3)}{\max(|w|, |h|)}, \qquad (6.36)$$

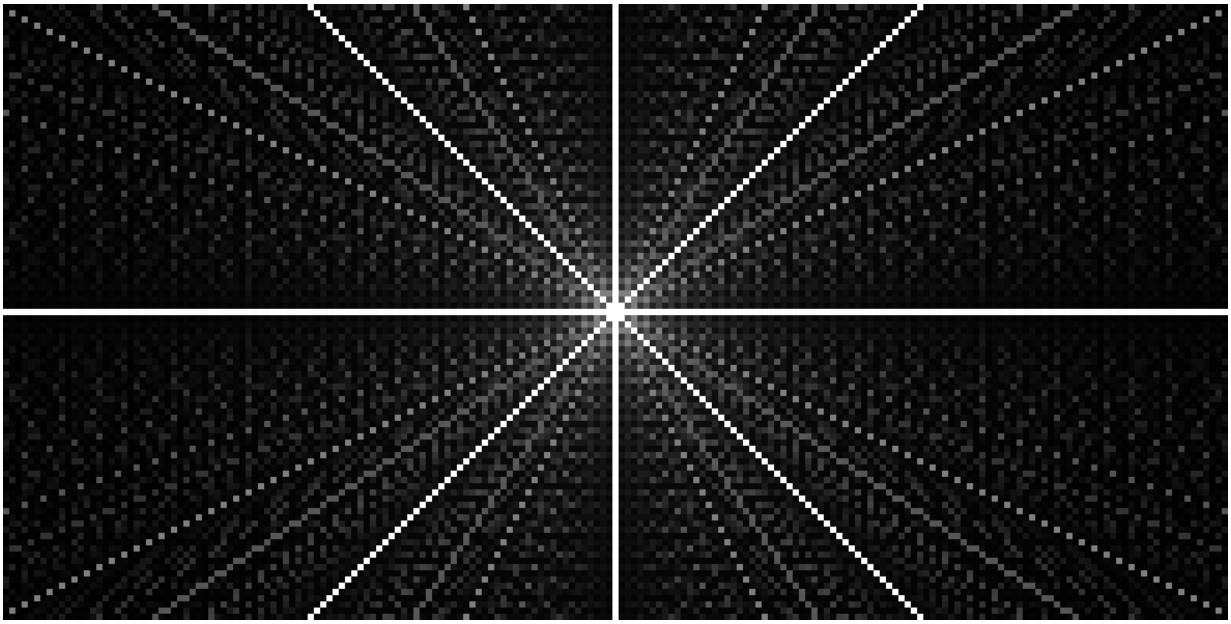


Figure 6.22: A heat map of the quality of aliased lines using ELDA-A. The centre of the image is $(0, 0)$. The intensity of pixel (w, h) represents the quality $Q_{\mathrm{A}}(w, h)$ of a line with width w and height h. The intensity value are normalized to the range $[0, 1]$.

where

$$g_1 = \gcd(A, B), \tag{6.37}$$

$$g_2 = \begin{cases} \gcd(A, B-1) & \text{if } A \bmod (B-1) \equiv 0,\ 1 < B < A, \\ \gcd(B, A-1) & \text{if } B \bmod (A-1) \equiv 0,\ 1 < A < B, \\ 1 & \text{otherwise,} \end{cases} \tag{6.38}$$

$$g_3 = \begin{cases} \gcd(A, B+1) & \text{if } A \bmod (B+1) \equiv 0,\ B < A,\ B < \max(w, h), \\ \gcd(B, A+1) & \text{if } B \bmod (A+1) \equiv 0,\ A < B,\ A < \max(w, h), \\ 1 & \text{otherwise.} \end{cases} \tag{6.39}$$

Figure 6.22 shows a heat map of $Q_{\mathrm{A}}(w, h)$. Some of the imperfect lines are now brighter because the revised measure increased their quality.

These measures of quality for both aliased and antialiased lines are useful for snapping endpoints of a line to maximize its quality. The idea is to pick a starting point p_1, an end point p_2, and a radius r. While keeping p_1 fixed, we search within a neighbourhood of p_2

of radius r to find another endpoint p_2' that is not too far from p_2 but improves the quality of the resulting line. In other words, if we denote the neighbourhood by $N_r(p_2)$, then we want to find p_2' that satisfies

$$Q(p_2') = \max_{\rho \in N_r(p_2)} Q(p_1, \rho) G(p_2, \rho), \tag{6.40}$$

where $Q(p_1, \rho)$ is the quality of the line from p_1 to ρ (it can be either aliased or antialiased), and $G_r(p_2, \rho)$ is the two-dimensional Gaussian function defined as

$$G_r\left((x_0, y_0), (x_1, y_1)\right) = \exp\left(-\frac{(x_1 - x_0)^2 + (y_1 - y_0)^2}{2(r/2.5)^2}\right). \tag{6.41}$$

The radius parameter r penalizes pixels far away from p_2, permitting a trade-off between line quality and proximity to endpoints.

Figure 6.23 illustrates the steps and the results of the line snapping. The red line is the original line we wish to draw with the top-left endpoint as p_1 and the bottom-right one as p_2. Figure 6.23a plots the line quality in a small neighbourhood around p_2, and Figure 6.23b shows the Gaussian values. In Figure 6.23c, we multiply the two values together and choose the brightest point (marked in green) as p_2'. Figures 6.23d and 6.23e are the pixelated lines drawn before and after snapping, respectively. The second line is of higher quality since it is a perfect line.

A line-drawing tool that allows snapping lets the user design isometric pixel art easily. Figure 6.24a shows a crudely drawn isometric cube, but with line snapping, it becomes much neater, as shown in Figure 6.24b. Right now our tool is interactive and the snapping occurs only for the line currently being drawn. For future work, we would like to extend the algorithm so that it can take an entire vector polygonal path as input and determine how to snap all the vertices simultaneously so that the resulting pixelation contains many high-quality line segments while remaining a faithful representation of the vector input.

6.5 Run-time Analysis

In this section, we analyze the performance of our algorithm and compare it to that of Bresenham's algorithm. Starting with ELDA-A, recall that it contains the follow steps:

1. Compute the set of units needed to construct the line.
2. Arrange them recursively in a bottom-up fashion.

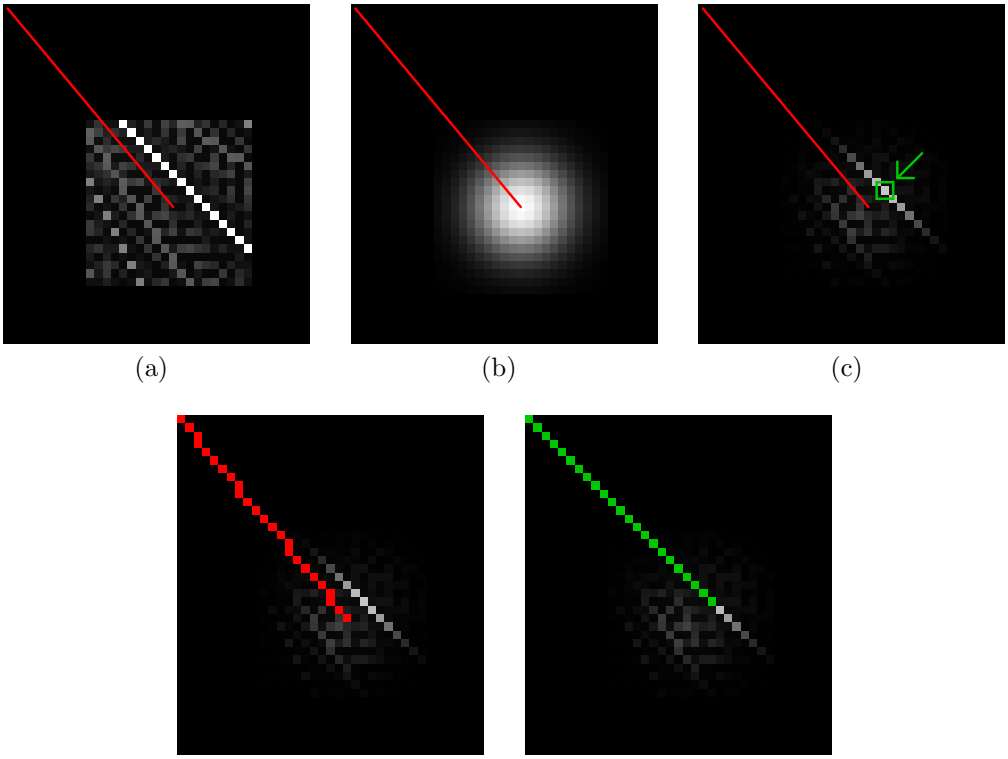
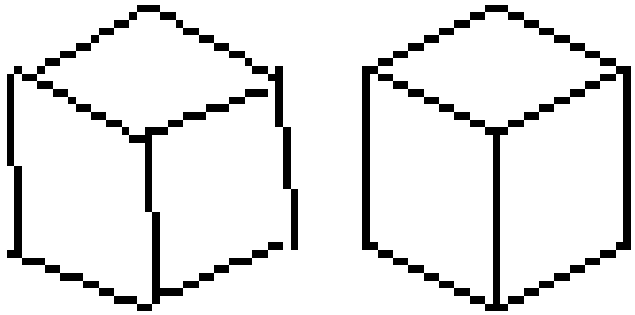


Figure 6.23: Line snapping based on quality measures: (a) values of Q_{A} in a small neighbourhood around the endpoint of the line, (b) 2D Gaussian values around the endpoint, (c) the product of Q_{A} and the Gaussian values around the endpoint, with the brightest point marked in green, (d) the pixelated line before snapping, and (e) the pixelated line after snapping.

3. Draw the pixels.

The third step—the drawing step—takes an amount of time proportional to the number of pixels, which is proportional to the length of the line. We will ignore it for the purpose of comparing the efficiency because, first, the number of pixels drawn is roughly the same in all line-drawing algorithms, and second, most line-drawing algorithms are so fast that the drawing step becomes the bottleneck. We are more interested in how quickly an algorithm can compute the set of pixels that need to be drawn.



(a) (b)

Figure 6.24: An isometric cube (a) before snapping and (b) after snapping.

The first step consists of simple $\mathcal{O}(1)$ calculations that take a negligible amount of time. The second step—the recursive step—is essentially the Euclidean algorithm, with the additional work of appending shorter line segments together to form longer line segments. Recall that the binary representation of the pixelated line segment constructed at Step i is given by

$$\mathbf{c}_i = \begin{cases} \{1\} & \text{if } i = -2, \\ \{0\} & \text{if } i = -1, \\ \left\lceil \frac{q_i}{2} \right\rceil \mathbf{c}_{i-1} + \mathbf{c}_{i-2} + \left\lfloor \frac{q_i}{2} \right\rfloor (-\mathbf{c}_{i-1}) & \text{if } 0 \geq i < \ell, \\ \left\lceil \frac{a_i}{2} \right\rceil \mathbf{c}_{i-1} + \left\lfloor \frac{a_i}{2} \right\rfloor (-\mathbf{c}_{i-1}) & \text{if } i = \ell, \end{cases} \tag{6.42}$$

where ℓ is the total number of steps. The recurrence relation suggests that each intermediate step performs q_i appends, and the last step performs a_i appends.

We want a data structure that appends multiple arrays efficiently. A linked list can append two arrays in $\mathcal{O}(1)$ time. However, it does not keep a copy of the original arrays, which is problematic if we want to append multiple copies of one array to another. If we simply use a regular dynamic array, then appending an array of size n to another takes $\mathcal{O}(n)$ time.[1]

6.5.1 Using the Binary Representation

If we cannot append the arrays quickly, perhaps we can find a shorter representation for pixelated lines so that the sizes of the arrays we are appending are smaller. Currently, our

[1]Our Java implementation uses a Vector, which is a type of dynamic array.

algorithm uses the binary representation to describe pixelated lines. Appending two lines is done by appending their respective binary representations. To determine the associated running time, we timed how long it takes for our algorithm (implemented in Java) to draw 500 lines of width w and height h, where w and h each range from 0 to 300. The resulting 300×300 array of times are plotted as a heat map, and scaled in order to use the full greyscale spectrum from black to white.

Figure 6.25a shows the heat map of the running times. The intensity of pixel (x, y) indicates how slowly our algorithm draws a line of width x and height y. The brighter the pixel, the more time it takes. The plot shows several distinctly bright lines that all have perfect slopes, which seems counterintuitive. These lines are the most highly structured and should be easy to draw as their pixel structures are simple to compute.

To make sense of Figure 6.25a, let us try to reproduce it by counting the operations in the actual algorithm. We can add up the sizes of all the appended arrays, and this produces the plot in Figure 6.25b. This plot looks different from Figure 6.25a because it shows that perfect lines take less time.

So what causes the bright lines in Figure 6.25a? It turns out that constructing a perfect line requires appending many short units together and each append has an overhead cost. Figure 6.25c shows the number of appends required to draw each line. If we factor in these overhead costs, the result we get (see Figure 6.25d) looks similar to the plot of the running time in Figure 6.25a.

This analysis shows that using the binary representation is still not enough to make the algorithm draw lines with high regularity faster. We need a different representation for pixelated lines.

6.5.2 Using the Compact Representation

The *compact representation* of a line is the run-length encoding of its binary representation. For example, Figure 6.26a shows a line drawn with orange and blue units. Above the line, we have the binary representation where 0 indicates an orange unit and 1 a blue unit. Below the line, we have the compact representation, which counts the length of the runs of alternating orange and blue units. To avoid ambiguity, we will always begin by counting the shorter unit (in this case, the orange one). If the line happens to start with the longer unit such as in Figure 6.26b, then the compact representation will have a leading zero.

With the compact representation, all perfect lines can be expressed concisely as an array of size 1 or 2. Other lines with semi-regular structures can be expressed more concisely as

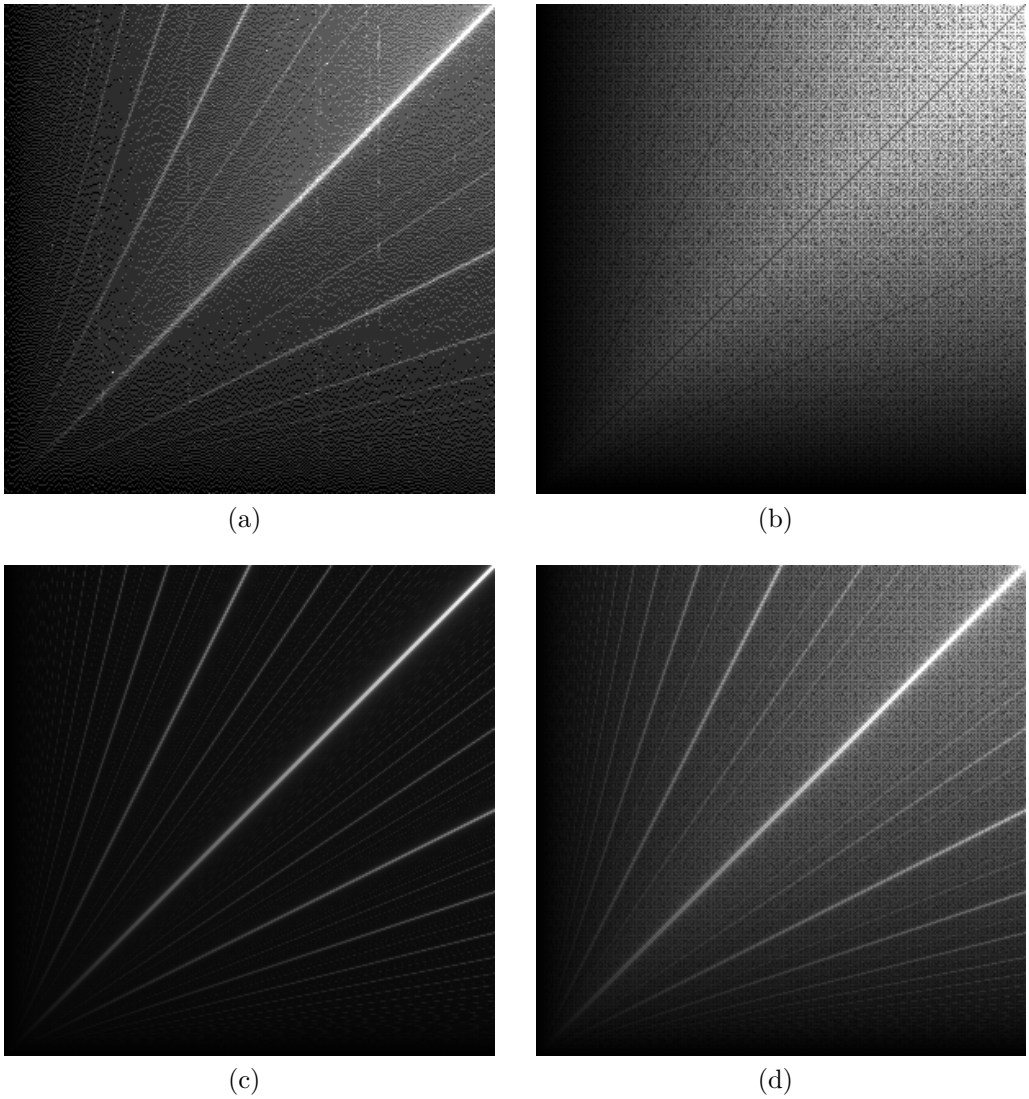
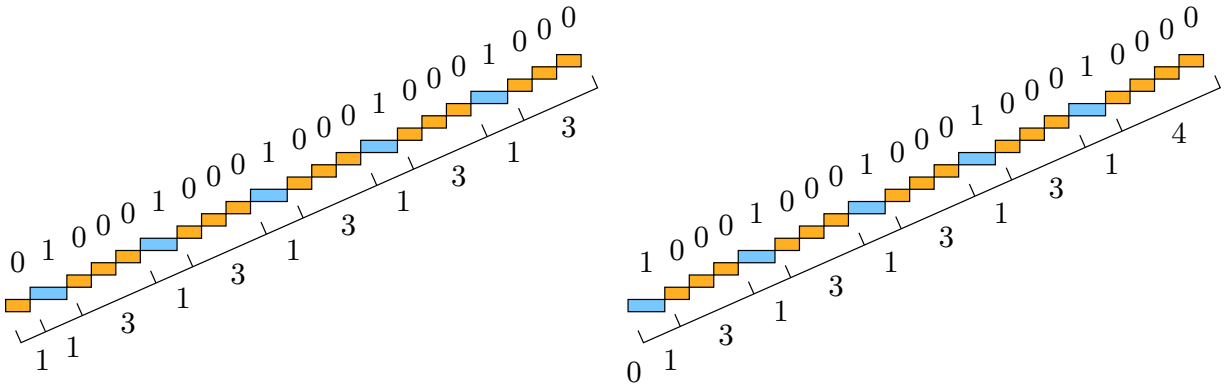


Figure 6.25: Run-time analysis of ELDA-A using binary representations: (a) running time of our implementation, (b) number of operations without considering overhead cost, (c) overhead cost from array appends, (d) number of operations considering overhead cost. All the values have been scaled to maximize contrast in the images.

well. Using this new representation, we timed how long it takes to draw lines of various sizes and plotted the heat map, as shown in Figure 6.27a. Now it has the feature we want,



(a) (b)

Figure 6.26: Compact representation of a pixelated line that starts with (a) a short unit represented by a 0, or (b) a long unit represented by a 1.

namely, that it takes the least amount of time to draw perfect lines and is faster than using the binary representation.

Again, we will compare the timing results with theoretical estimates. Adding the sizes of all the appended arrays in the algorithm produces the plot in Figure 6.27b. This plot contains most of the characteristics of the running time plot, except for the finer details such as the brighter lines. An even simpler estimate of the running time is the length of the line's compact representation, as shown in Figure 6.27c. The plot explains the general trend in the timing plot with the smooth gradients.

As for the lower-level details such as the bright lines, they can be explained by how deep the recursion goes. Figure 6.27d shows a plot of the number of iterations it takes to draw each line. Here, the darkest lines all have perfect slopes because they take one iteration, but this plot contains information about other lines as well. For example, lines whose slopes are halfway between two perfect slopes are also darker because they take only two iterations. Basically, the higher the pixel pattern regularity, the lower the number of iterations.

The brightest lines (i.e., the lines that take the longest to draw) are the least regular ones, and their slopes are all related to the Golden Ratio φ. These slopes are rational numbers closest to $\pm(\varphi + k)$ or $\pm\frac{1}{\varphi+k}$ for some nonnegative integer k. In Figure 6.27d, the coordinates of the red points are successive Fibonacci numbers. These points lie close to the line $y = \varphi x$ and they indicate places where the algorithm is particularly slow. The reason is that, when the Euclidean algorithm is applied to two successive Fibonacci numbers, all

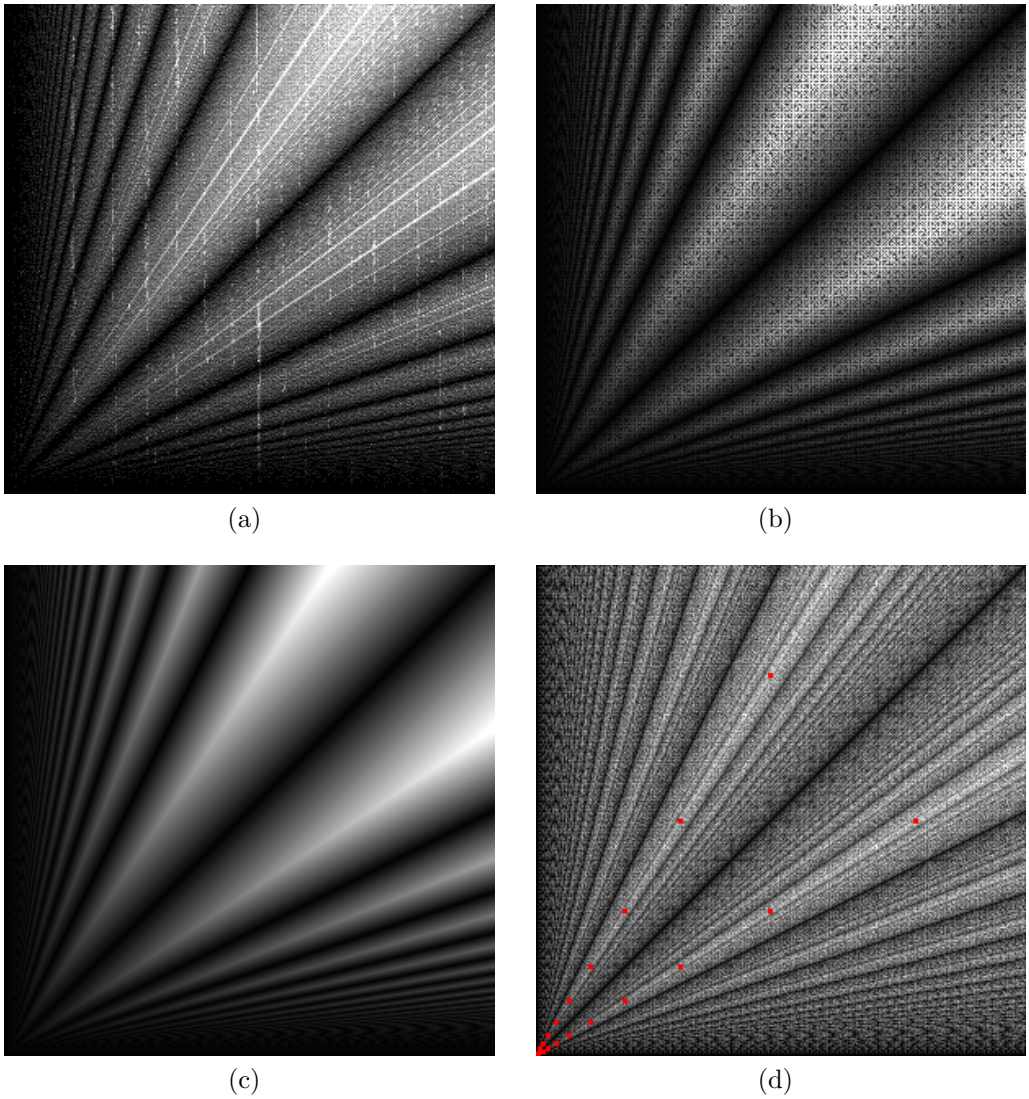
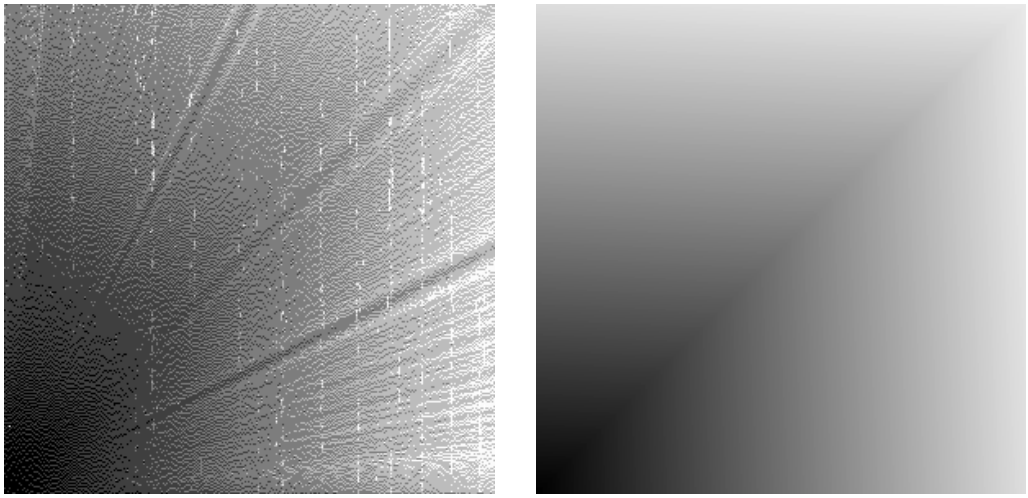


Figure 6.27: Run-time analysis of ELDA-A using compact representations: (a) running time of our implementation, (b) number of operations, (c) lengths of the compact representations of the lines, and (d) number of iterations needed to draw the lines (the red dots have coordinates that are successive Fibonacci numbers). All the values have been scaled to maximize contrast in the images.

the quotients are 1, and therefore it takes the maximum number of iterations to complete the recursion.



(a) (b)

Figure 6.28: Run-time analysis of Bresenham's line-drawing algorithm: (a) running time of our implementation, and (b) number of operations.

6.5.3 Comparison to Bresenham's Line Algorithm

For the sake of comparison, we also performed the same analysis on Bresenham's algorithm. Figure 6.28a shows the timing plotted as a heat map. The gradient outward from the origin suggests the algorithm runs in linear time with respect to the length of the line it draws. There are also dark lines emanating from the origin, suggesting that lines with slope $\frac{1}{2}$, 1 and 2 are drawn more quickly than others.

In Figure 6.28b, we plotted the theoretical times, calculated by assuming each addition takes $\mathcal{O}(n)$ time and each multiplication takes $\mathcal{O}(n^2)$ time, where n is the number of digits, which is 32 in our implementation. Although this plot captures the general trend of the actual timing, it does not explain the darker lines. The minor discrepancy could be due to hidden run-time costs associated with certain arithmetic operators in our implementation, which are not being captured by the simplistic counting of additions and multiplications.

In summary, the analysis shows that when pixelating a line with ELDA-A, the running time is proportional to the length of the line's compact representation. In other words, lines with high pixel regularity are drawn faster than those with low pixel regularity. Hence, if we use the quality measure to select only good lines to draw, then our algorithm can be very efficient. In contrast, the running time of Bresenham's algorithm scales linearly with the length of the line and therefore does not take advantage of the pixel regularity

to speed up the calculations. However, despite this advantage, ELDA-A is currently not as fast as Bresenham's because it has more overhead costs and we have not optimized our implementation. For future work, we would like to create a more efficient implementation that can be used for general purpose line rasterization.

Chapter 7

Drawing Angles

In the previous chapter, we discussed how to draw lines. In this chapter, we will describe how to draw angles formed by aliased lines in a way that meets isometric pixel art standards. We first introduce some terminology. As shown in Figure 7.1, an angle is formed by two edges meeting at a vertex. Edge 1 and Edge 2 are defined such that the counterclockwise angle measure from Edge 1 to Edge 2 is no greater than 180°. When an edge is rasterized, the pixel and pixel span closest to the vertex are called the *initial pixel* and *initial span* of that edge, respectively. When the two edges join to form a pixelated angle, the initial spans may overlap and the overlapping pixels are called the *intersection* or *intersected pixels*.

For consistency, our illustrations in this chapter will use green for Edge 1, red for Edge 2, and yellow for their intersection. This colouring helps visualize how two edges join. Sometimes we will also draw both edges with the same colour in order to show artifacts more clearly.

Now consider the four pairs of edges in Figure 7.2. For each pair, we create an angle by overlapping the initial pixels. In each case, the artifacts are marked in purple. Angles A and B both have L-shaped corners, while Angle C has several pixel clusters. As discussed in Section 3.1.2, L-shaped corners are undesirable because they stand out and affect our ability to perceive the entire angle as one object. In Angle C, the pixel clusters are formed because the two edges are too close together and the pixel-level details in each edge are lost as result. As for Angle D, joining the two initial spans creates a merged pixel span that may be too long.

The angle artifacts shown in Figure 7.2 are only conjectured. Unlike the other topics in pixel art, we could not find any tutorials that specifically discuss angle drawing. To understand how pixelated angles are drawn, we did a preliminary survey of pixel art

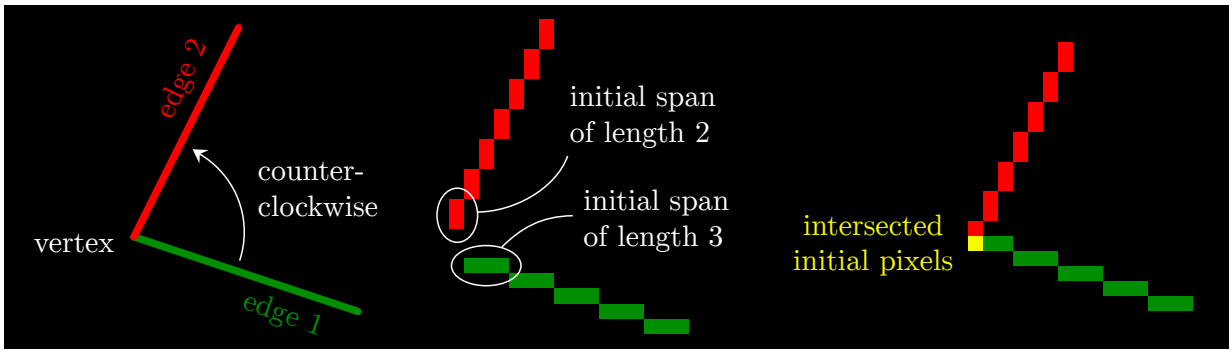


Figure 7.1: Two edges meet at a vertex to form an angle . The counterclockwise angle measure from Edge 1 to Edge 2 is at most 180°. When pixelated, the pixel and span closest to the vertex is called the initial pixel and initial span respectively. When joined together, overlapping pixels form an intersection.

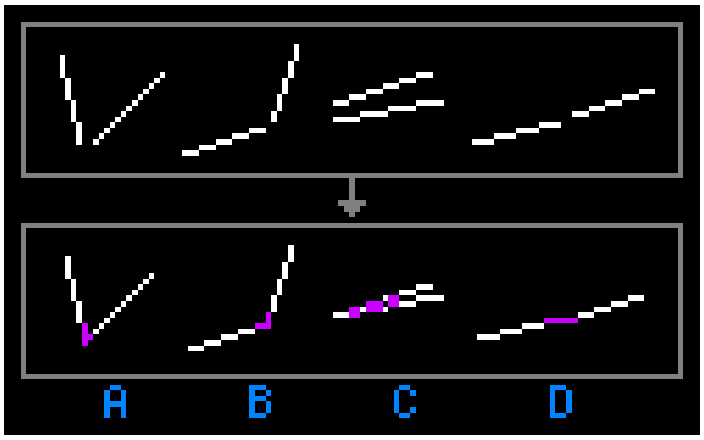
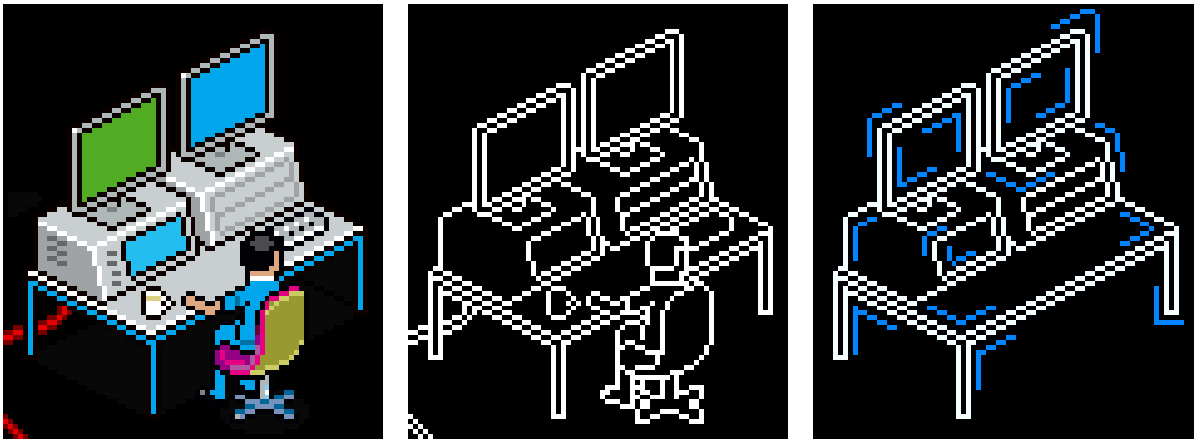


Figure 7.2: Artifacts in naïvely drawn angles (in purple): (a,b) L-shaped corners, (c) pixel clusters, and (d) spans that are too long.

compositions and then contacted pixel artists for clarification on certain cases. From the collected data, we devised a set of design principles on which our angle-drawing algorithm is based. Finally, we asked pixel artists to evaluate the results of our algorithm.



(a) (b) (c)

Figure 7.3: (a) An example of eBoy's isometric pixel art, (b) the line art, and (c) some angles extracted from it. Used with permission.

7.1 Angles in Isometric Pixel Art

For the preliminary survey of angles in pixel art, we studied isometric pixel art compositions by eBoy. We chose their work because they have a large portfolio of high-quality work and they specialize in isometric pixel art, which often contains many angles made by straight lines. We collected several pixel art compositions from their website and manually extracted the angles.

To demonstrate the angle extraction process, consider the computer scene in Figure 7.3a. First we first extract the line art (as shown in Figure 7.3b in white). There are some extraneous details such as the person and the mug that we remove and the missing regions are then filled in as much as we can using information from the background. We also remove outlines that are not straight lines since we are only interested in angles with straight edges. After cleaning up the line art, we identify all the angles, a subset of which is shown in Figure 7.3c in blue.

Unfortunately, not all examples are as clear-cut as the ones in Figure 7.3. We did not extract the angles through automatic means because there are many ambiguities that are difficult to resolve. Figure 7.4 shows some of the problems we encountered. In Figure 7.4a, the circled portion appears to contain one rounded-looking angle, but there are actually two angles formed by three edges. Figure 7.4b contains an angle from a filled region, but we want outlined angles instead. Figure 7.4c shows multiple angles created by several lines

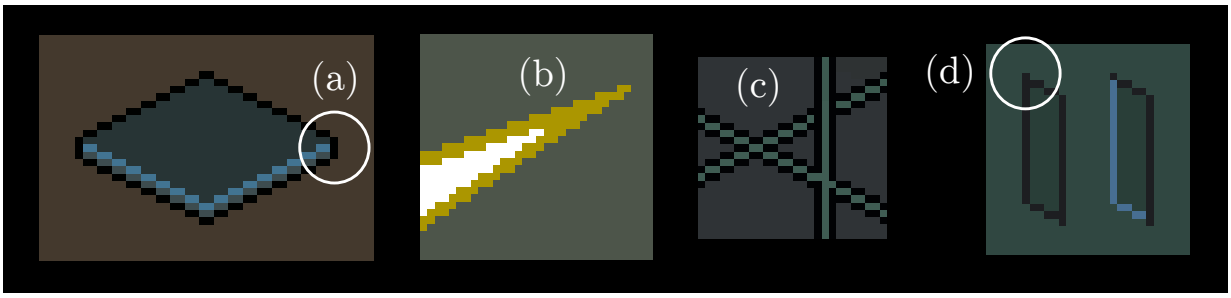


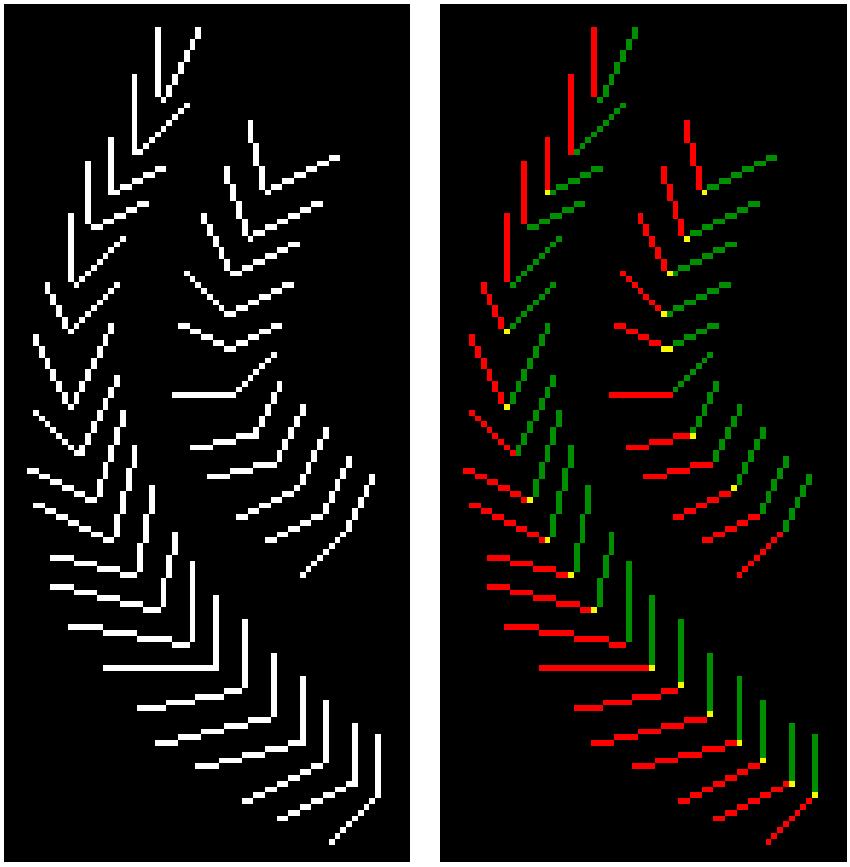
Figure 7.4: Problems in angle extraction: (a) rounded corners, (b) angles that are not outlined, (c) angles created by multiple lines intersecting, and (d) L-shaped corners suggesting occlusion. Used with permission.

intersecting. These angles are most likely not drawn in the best way possible because the intersecting lines create other constraints. In Figure 7.4d, we circled an angle that contains an L-shaped corner; usually an L-shaped corner is considered an artifact, but in this case it may be used to suggest occlusion between two edges.

In some of these cases, it is not clear how to extract the angles. Even if they are extracted, since they are used in a particular setting, these angles may not be well-suited for general purpose angle drawing such as drawing a simple polygon. In our angle extraction process, we try to ignore these more difficult cases. To ensure that the angles we extracted are suitable for general purpose angle drawing, we conducted a survey (see Section 7.2 for more detail) to get additional information from eBoy directly.

Figure 7.5a shows a set of angles we extracted from six different eBoy drawings. Notice that, given two edges, the angle formed may be drawn in different ways. Some of the angles may be interpreted as two edges intersecting at their initial pixels or initial spans, while others do not appear to intersect at all. The interpretations are not unique but in Figure 7.5b we coloured the edges to show how some edges appear to intersect while others do not.

All these angles adhere to some basic rules. First, for each angle, the initial pixels of the two edges are always within one pixel horizontally and vertically of each other. Figure 7.6a shows an angle violating this rule. Second, if two edges intersect, their initial pixels are always in the intersection. Figure 7.6b and 7.6c show two violations. Third, every pixel in the angle is in the trajectory of at least one of the edges. The trajectory of an edge is its extension to infinity in both directions. In Figure 7.6c, the trajectories of the red and green edges are drawn in dark red and dark green respectively. In this case, the angle violates the third rule because the yellow pixel is outside of both trajectories. Fourth, each angle



(a) (b)

Figure 7.5: (a) Angles extracted from isometric pixel art, (b) each drawn as two edges (red and green) possibly with intersections (in yellow).

contains at most one L-shaped corner and none of them contain pixel clusters. Finally, the vertex of an acute angle is almost always represented with a single pixel (as shown in Figure 7.6e) and rarely with a span of length greater than 1 (see Figure 7.6f).

7.2 Angle Survey

Using the basic rules we developed from analyzing the extracted angles, we can narrow down the set of good angles, but these rules alone are still not enough to develop an effective

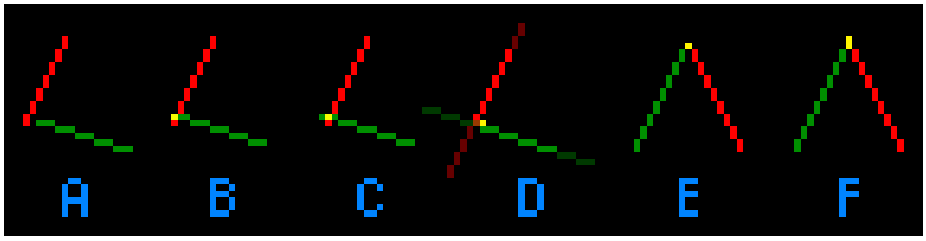


Figure 7.6: Angles not found in eBoy's pixel art: (a) disconnected edges, (b) intersection not containing one of the initial pixels, (c) intersection not containing either initial pixel, and (d) a pixel (in yellow) not contained in the trajectory of either edges. In an acute angle, the vertex is mostly represented with (e) a single pixel, and rarely with (f) a span of length greater than 1.

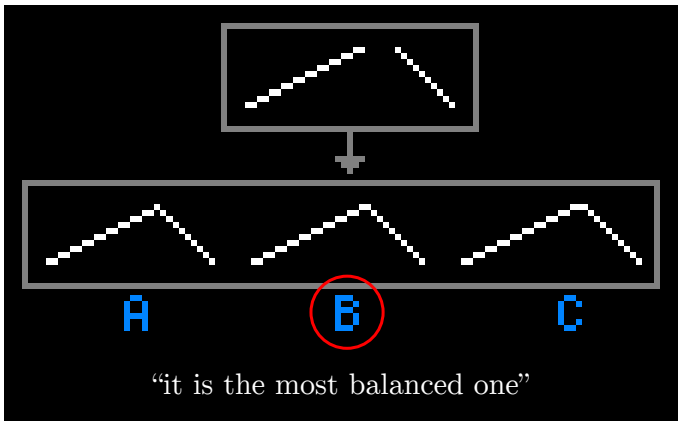


Figure 7.7: eBoy's angle choice and comment response for Question 1 of the angle survey.

algorithm. To gather more information, we created a survey that contains seven angles, each drawn in several different ways following the basic rules. We asked the eBoy artists to complete this survey; for each angle, they were asked to choose the best-looking angle and explain their choice. We will show their responses and try to interpret them using our terminology.

For Question 1 (see Figure 7.7), eBoy chose B because "it is the most balanced one." We interpret this comment to mean that the merged span created from the two initial spans is neither too long nor too short.

For Question 2 (see Figure 7.8), they chose D since there is "no accumulation of pixels where the lines join." We believe this comment is referring to the lack of pixel clusters and L-shaped corners near the vertex.

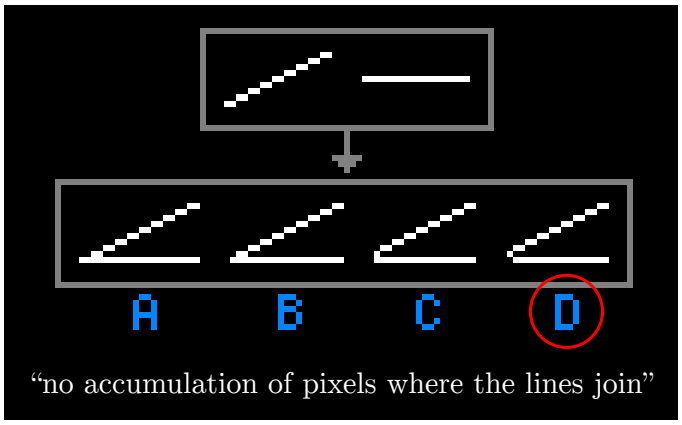


Figure 7.8: eBoy’s angle choice and comment response for Question 2 of the angle survey.

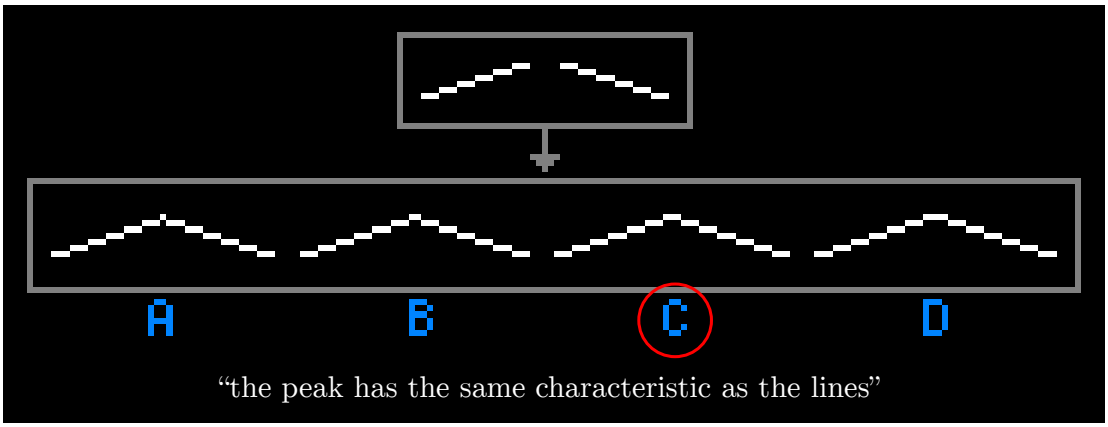


Figure 7.9: eBoy’s angle choice and comment response for Question 3 of the angle survey.

For Question 3 (see Figure 7.9), C was chosen because “the peak has the same characteristic as the lines.” The peak is most likely referring to the merged span at the vertex, and for Option C, the merged span has length 3, like the other spans in the two edges.

For Questions 4 and 5 (see Figures 7.10 and 7.11), the choices were again based on the lack of L-shaped corners and pixel clusters.

For both Questions 6 and 7 (see Figures 7.12 and 7.13), eBoy gave two possible answers. In Question 6, C was chosen because it is a “good mix of B and D”—most likely referring to the length of the merged span, and D was their secondary choice. In Question 7, B was chosen due to “the peak [having] the same characteristic as the line,” referring to the

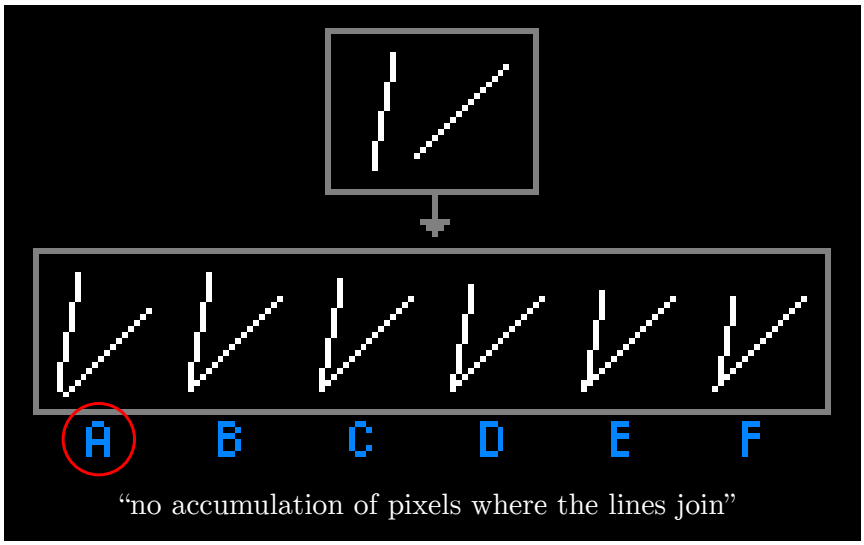


Figure 7.10: eBoy’s angle choice and comment response for Question 4 of the angle survey.

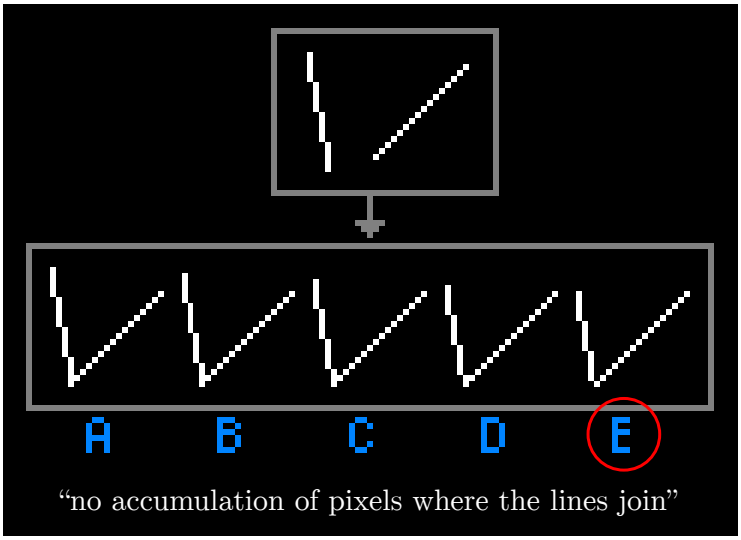


Figure 7.11: eBoy’s angle choice and comment response for Question 5 of the angle survey.

initial spans having the same lengths as the spans in the edges; D was the secondary choice because it also “looks great.” Our algorithm will be based on their primary choices.

To summarize, the angle survey shows that eBoy focuses on three main criteria when deciding on how to draw pixelated angles:

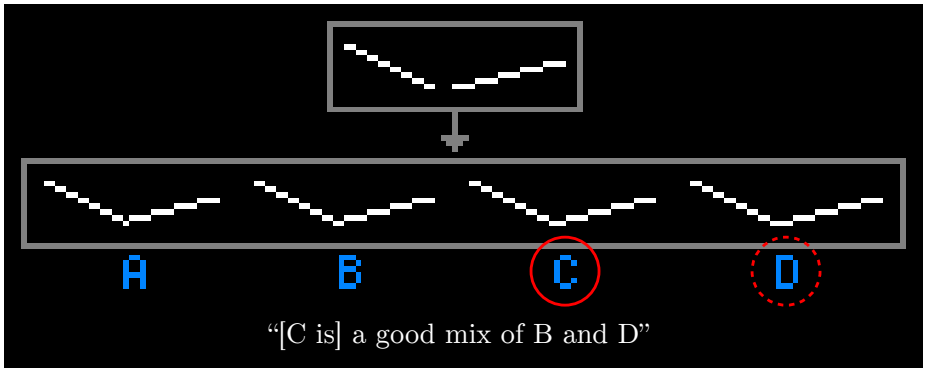


Figure 7.12: eBoy's angle choices and comment response for Question 6 of the angle survey. C is the primary choice and D is the secondary choice.

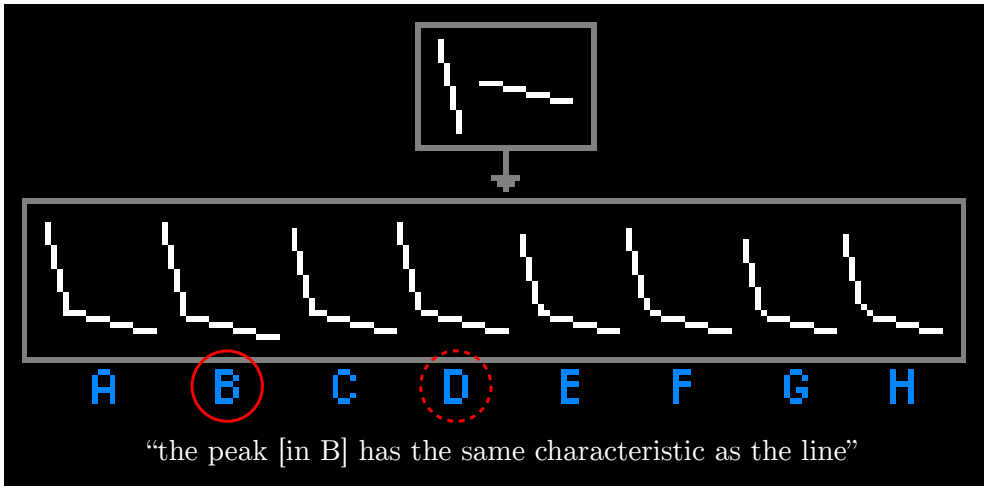


Figure 7.13: eBoy's angle choices and comment response for Question 7 of the angle survey. B is the primary choice and D is the secondary choice.

1. Avoid pixel clusters.
2. The merged span should be neither too long nor too short.
3. If possible, use only spans found in the edges.

The first point is the most important since we did not find any exceptions in eBoy's drawings. For the second point, we will define later what a good length for a merged span

should be. As for the third point, we will follow it only if it does not violate the second point.

7.3 Angle Classification

The preliminary observations and the angle survey show that there are different considerations for different types of angles. For example, avoiding pixel clusters is the main concern for narrow angles, but not for other types of angles. We want to categorize the angles by how they behave when they are pixelated.

We first attempted to categorize the angles by their angle measures, but soon found that angles with the same measures can sometimes behave fundamentally differently. After further exploration, we discovered a better classification scheme in terms of octants and quadrants which will be used later to describe our angle-drawing algorithm.

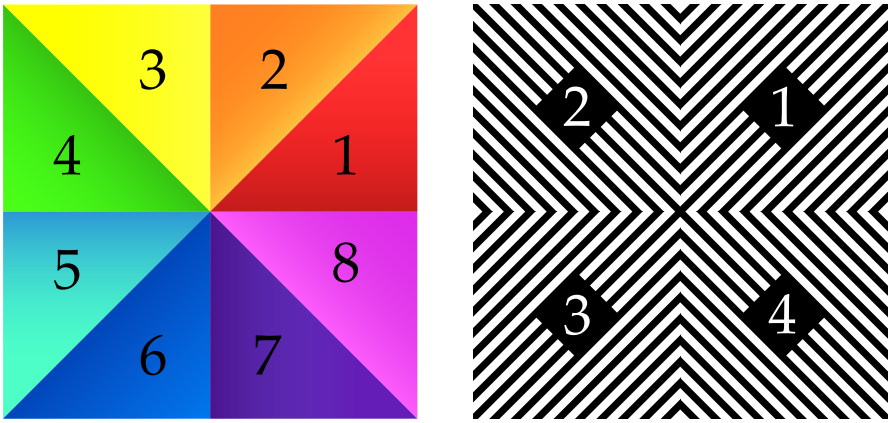
An angle is composed of two edges, each belonging to one of eight octants as shown in Figure 7.14a. There are also four quadrants, as shown in Figure 7.14b. We classify pixelated angles by octant ranges and quadrant ranges. The *octant range* of angle is the number of octants covered by the interior of that angle. Similarly, the *quadrant range* of angle is the number of quadrants it covers. Figure 7.15 shows examples of angles with octant ranges of 1 through 5, and quadrant ranges of 1 through 3. It is important to keep track of the variables separately because the octant range of an angle does not determine the quadrant range.

7.4 Angle-Drawing Algorithm for Perfect Slopes

We will start by describing a simple version of our algorithm that involves only edges with perfect slopes.

7.4.1 Phase Shifting and Angle Widening

First let us consider angles with octant ranges up to 3. Since these angles are relatively narrow, we want to make sure not to introduce pixel clusters. To avoid pixel clusters, we phase shift the edges so that the angle appears slightly wider. Phase shifting basically involves changing the length of the initial span. Figure 7.16a shows an example of how it



(a) (b)

Figure 7.14: (a) Eight octants (shown in different colours) and (b) four quadrants (indicated by different line orientations).

can be used to widen an angle. In this example, Edge 1 (in green) has slope 1 and Edge 2 (in red) has slope -5. There is one way to draw Edge 1 since its initial span must be of length 1. However, there are five ways to draw Edge 2 because its initial span can range from 1 to 5. In this case, 1 is the *minimum span length* and 5 is the *maximum span length*. Using the five different representations, we can create five angles by overlapping the initial pixels of the two edges. As the initial span length of Edge 2 decreases, the angle appears wider. In this case, Angle E is the best choice according to Question 5 from the angle survey.

Phase shifting makes an angle appear wider because the edges appear to be translated away from each other. For example, in Figure 7.16a, as the initial span of Edge 2 (in red) shortens, it appears to be translated to the left, away from Edge 1 (in green). The way we defined Edges 1 and 2, translating them away from each other means shifting Edge 1 in the clockwise direction while shifting Edge 2 in the counterclockwise direction, as shown in Figure 7.16b. Basically, to shift an edge in the clockwise direction, we must minimize the initial span length if the edge is in an even-numbered octant and maximize the length if the edge is in an odd-numbered octant. Conversely, to shift it in the counterclockwise direction, we minimize the initial span length if the edge is in an odd-numbered octant and maximize it if the edge is in an even-numbered octant.

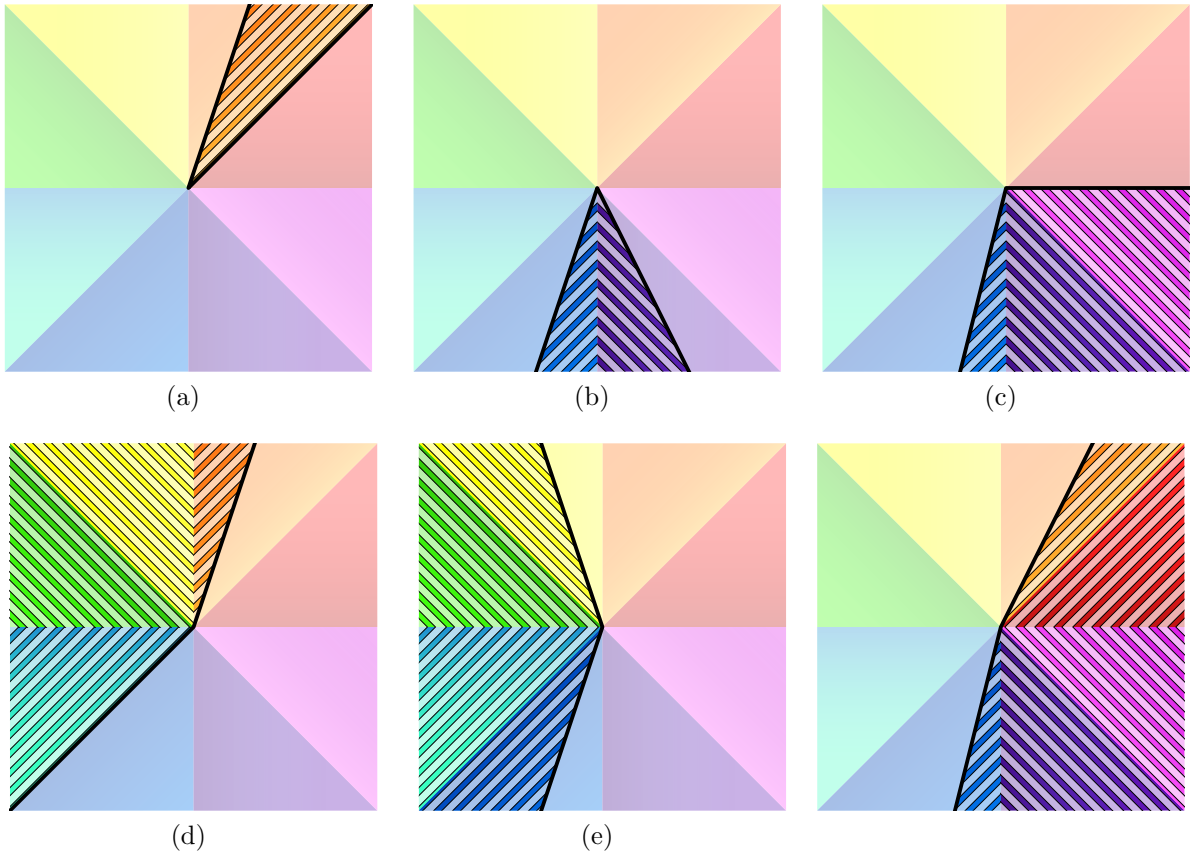
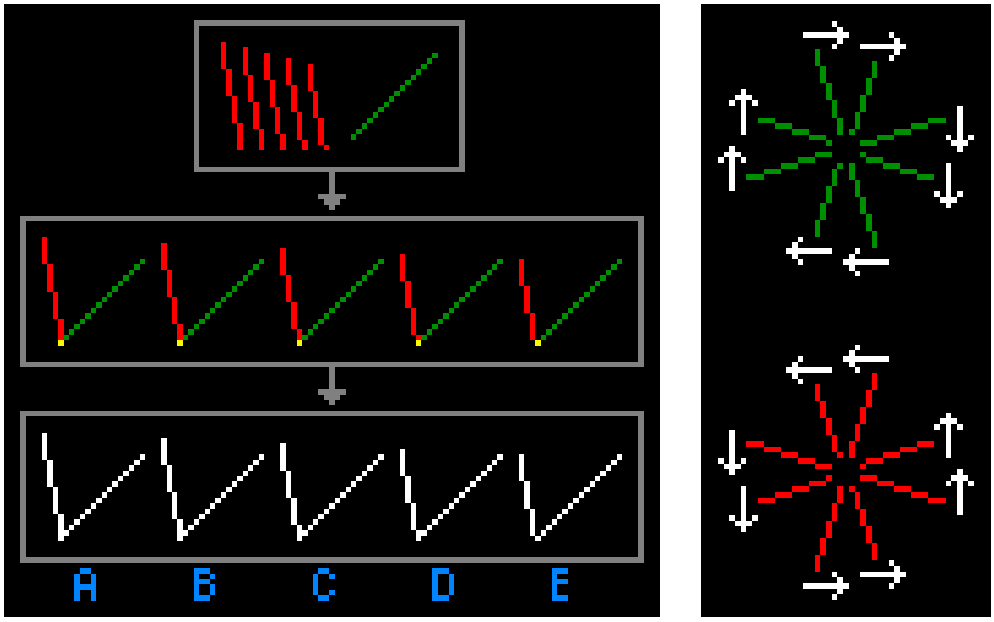


Figure 7.15: For each angle, its octant range and quadrant range are respectively (a) 1 and 1, (b) 2 and 2, (c) 3 and 2, (d) 4 and 3, (e) 4 and 2, and (f) 5 and 3.

7.4.2 Edge Offsetting

For narrower angles—those with octant ranges of 1—angle widening ameliorates but does not quite eliminate the pixel cluster problem. In Figure 7.17, the angles in the middle row are drawn using only the angle-widening method and they still contain pixel clusters. To eliminate pixel clusters, we need to widen the angles further by slightly offsetting the edges as shown in the bottom row of Figure 7.17.

Figure 7.18 shows eight angles with octant ranges of 1 in different octants. The arrows indicate the directions in which to offset the edges. The offset directions depend on which quadrant the angle is in. By offsetting the edges, the resulting angles are guaranteed to be free of pixel clusters.



(a) (b)

Figure 7.16: (a) A line of slope -5 has five different phases. Of the five angles produced, Angle E is the best-looking according to the angle survey. (b) To widen an angle, shift Edge 1 (in green) in the clockwise direction and Edge 2 (in red) in the counterclockwise direction.

7.4.3 Merging Truncated Spans

The remaining angles have octant ranges of at least 4, and their quadrant ranges are either 2 or 3. We will first discuss those with quadrant ranges of 2. Drawing these angles involving joining two initial spans to formed a *merged span*. According to the angle survey, the quality of the angle drawn depends highly on the length of the merged span.

Figure 7.19 shows three angles from the angle survey that were chosen by eBoy as best-looking. These examples suggest that the length of the merged span L_m should be calculated as

$$L_m = \left\lceil \frac{L_1 + L_2}{2} \right\rceil, \tag{7.1}$$

where L_1 and L_2 are the initial span lengths of Edge 1 and Edge 2 respectively.

Let us consider angle construction as a two-step process:

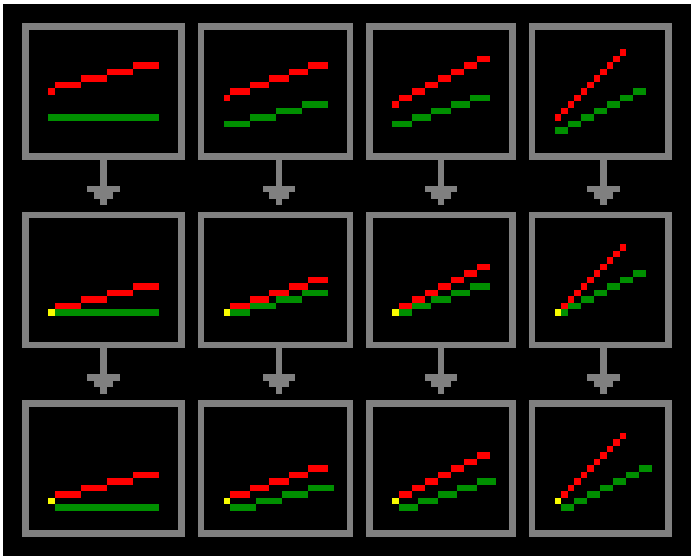


Figure 7.17: For angles with octant ranges of 1, angle widening is not enough to remove pixel clusters. The middle row shows angles drawn with only angle widening. The bottom shows angles drawn with both angle widening and edge offsetting.

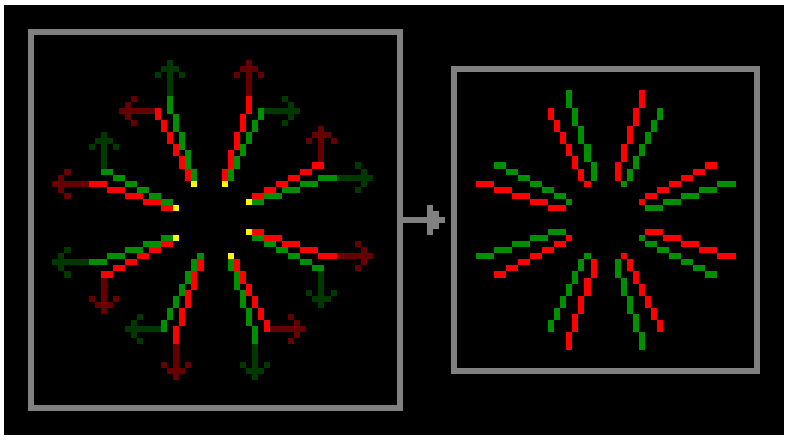


Figure 7.18: To remove pixel clusters in angles with octant ranges of 1, offset the two edges in the directions indicated by the arrows.

1. Truncate the initial spans and let L_1' and L_2' be the new initial span lengths.

2. Intersect the two truncated spans to form the desired merged span of length L_m.

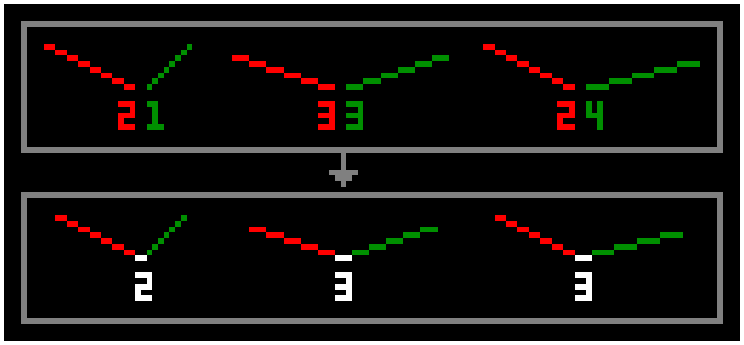


Figure 7.19: According to the angle survey, joining the spans in the top row should yield the merged spans in the bottom row.

The intersected pixel indicates where the vertex is, and knowing the position of the vertex in the pixelated angle helps us position the angle properly. For L'_1 and L'_2, we want them to satisfy

$$\frac{L'_1}{L'_2} = \frac{L_1}{L_2}, \tag{7.2}$$

$$L_m = L'_1 + L'_2 - 1. \tag{7.3}$$

Solving this system of equations gives us

$$L'_1 = (L_m + 1) \cdot \frac{L_1}{L_1 + L_2}, \tag{7.4}$$

$$L'_2 = (L_m + 1) \cdot \frac{L_2}{L_1 + L_2}. \tag{7.5}$$

L'_1 and L'_2 are rounded to the nearest integers.

Figure 7.20 shows some angles created by this method of merging truncated spans. Angles A, B and C are identical to the ones chosen by eBoy in the angle survey. The second row shows that $L_m = \left\lceil \frac{L_1+L_2}{2} \right\rceil$, and the last row shows that $L'_1 + L'_2 - 1 = L_m$.

7.4.4 Join Edges without Intersection

The last type of angles is those with quadrant ranges of 3. These angles are the easiest to draw because they are wide enough that pixel clusters are not an issue and they also do not create merged spans. We can simply draw the two edges without any phase shifting (i.e.,

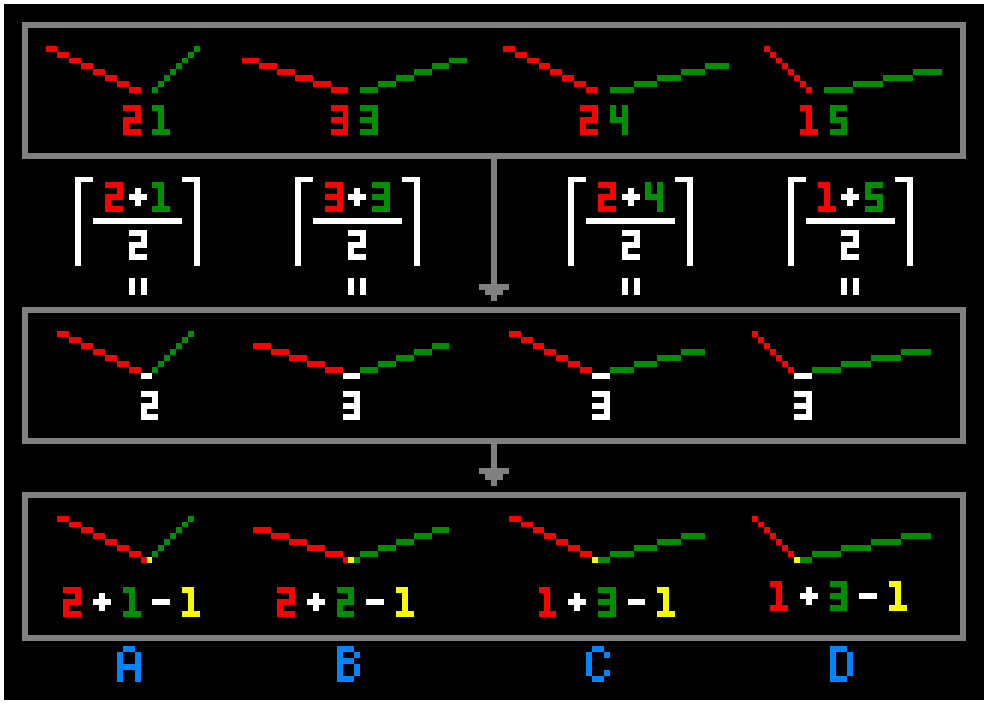


Figure 7.20: Four examples of how to form merged spans from truncated spans.

the initial spans are at their maximum lengths) and then join them without intersection. Figure 7.21 shows the angles produced by this method.

To summarize, our algorithm takes the slopes of the two edges as input, computes the octant range and the quadrant range of the angle formed, and then splits the angle-drawing process into the following four cases:

1. If the octant range is 1, then apply angle widening and edge offsetting (see Sections 7.4.1 and 7.4.2).

2. If the octant range is 2 or 3, then apply angle widening (see Section 7.4.1).

3. If the octant range is greater than 3 and the quadrant range is 2, then truncate and merge the initial spans (see Section 7.4.3).

4. If the quadrant range is 3, join the edges without intersection (see Section 7.4.4).

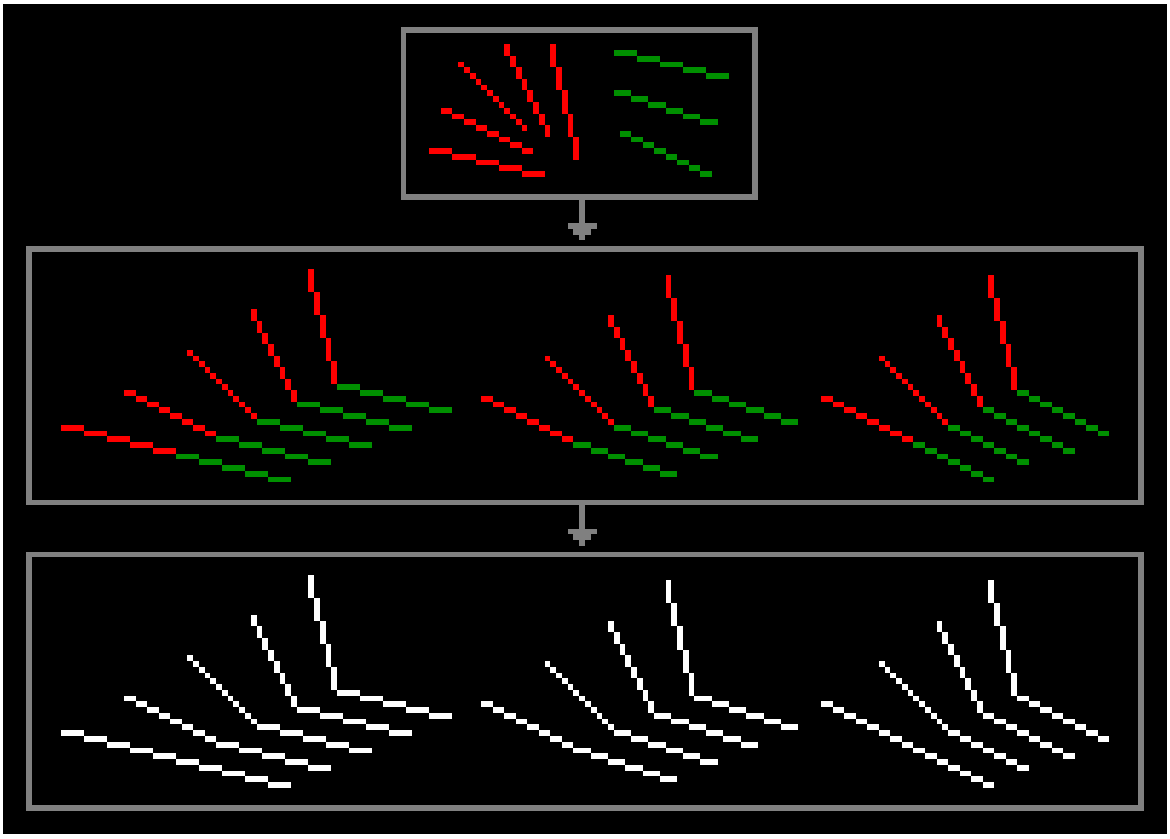


Figure 7.21: Algorithm results for angles with quadrant ranges of 3. The angles are created by joining the two edges without intersection.

7.4.5 Results and Evaluation

For evaluation, we compared the results of our algorithm to those of a naïve method. The naïve method draws an angle by drawing both edges without any phase shift, and then joining them by intersecting their initial pixels. We applied these two algorithms to a large set of angles formed from edges with perfect slopes. The results are shown in Figures F.1 and F.2 in Appendix F, since they are too large to display here.

We showed these two sets of angles to the eBoy artists for feedback. They agreed that our results are much better looking than the naïvely constructed angles. They noticed that for narrow angles, our representation does not look as sharp as the naïve representation. Regarding this difference, eBoy said that they would occasionally prefer the naïve representation if the sharpness of the angle is crucial to the composition, but in general,

Figure 7.22: (a) level-1 lines, (b) level-2 lines, (c) level-3 lines, and (d) all of above combined. The set of lines considered have slopes between $\frac{1}{4}$ and 4 inclusive.

the results of our algorithm are an effective characterization of their approach to angle drawing.

7.5 Angle-Drawing Algorithm for Imperfect Slopes

So far, our algorithm only applies to angles composed of perfect lines since they are the most common type of lines found in isometric pixel art. Our algorithm can be extended to imperfect lines, but we first introduce a method for classifying lines according to their slopes.

7.5.1 Line Slopes and Levels

Suppose we have an irreducible fraction $\frac{a}{b}$. If a line has this slope, then we define its *level* to be $\min(|a|, |b|)$. Figure 7.22 shows examples of level-1, level-2, and level-3 lines in the first quadrant whose slope values are in the interval $[\frac{1}{4}, 4]$. In this example, the slope of a level-1 line is either an element of or the reciprocal of an element of the set $\pm\{1, 2, 3, 4, \ldots\}$; the slope of a level-2 line is either an element of or the reciprocal of an element of the set $\pm\{\frac{3}{2}, \frac{5}{2}, \frac{7}{2}, \ldots\}$; the slope of a level-3 line is either an element of or the reciprocal of an element of the set $\pm\{\frac{4}{3}, \frac{5}{3}, \frac{7}{3}, \frac{8}{3}, \ldots\}$.

Roughly speaking, the level of a line indicates how irregular its span patterns are. A level-1 line is a perfect line with only one type of span. A level-2 line consists of alternating spans. A level-3 line has an even more complex pattern of alternating spans. In eBoy's

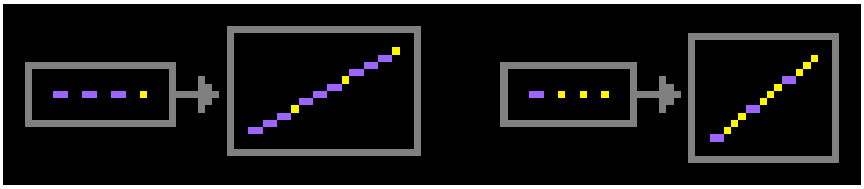


Figure 7.23: A line of slope $\frac{4}{7}$ is formed by three length-2 spans and one length-1 span. A line of slope $\frac{4}{5}$ is formed by three length-1 spans and one length-2 span.

drawings, level-1 lines are the most common, but level-2 and level-3 lines are occasionally used as well if the drawings require the use of certain imperfect lines. In Figure 7.22, we superimpose all these level-1, level-2 and level-3 lines to show that they provide fairly good coverage of the first quadrant; this suggests that using lines of the first three levels may be sufficient for most pixel art compositions.

When we describe the angle-drawing algorithm for imperfect slopes, we are really referring to angles made up of low-level lines. The algorithm is simply a generalization of the angle-drawing algorithm for perfect slopes, but slightly more complex due to the more complex pixel patterns in the lines. For our angle-drawing algorithms, we will focus on using lines up to level 3.

7.5.2 Algorithm Details

When dealing with imperfect slopes, each line is composed of two types of spans—a short one and a long one—that differ by one pixel in length. The pattern always consists of either a short span followed by several long ones, or a long span followed by several short ones, as shown in Figure 7.23. To keep the calculations simpler, we always default to starting a line with a long span, as shown by the lines in Figure 7.23. Then if we need to phase shift any edge, we simply change the length of its initial span.

Earlier, we defined the maximum span length of a perfect line to be the length of its spans. Now, for an imperfect line, the maximum span length refers to the length of the longer of the two spans. For example, the maximum span length of a line of slope $\frac{4}{3}$ is 2 since it consists of length-1 and length-2 spans.

Let us start with angles with octant ranges of 1. Consider the example in Figure 7.24 in which Edge 1 (in green) consists of length-1 and length-2 spans and Edge 2 (in red) contains length-2 and length-3 spans. Applying angle widening minimizes Edge 1's initial span length to 1 and maximizes Edge 2's initial span length to 3. Then the edges are offset slightly to ensure no pixel clusters are formed.

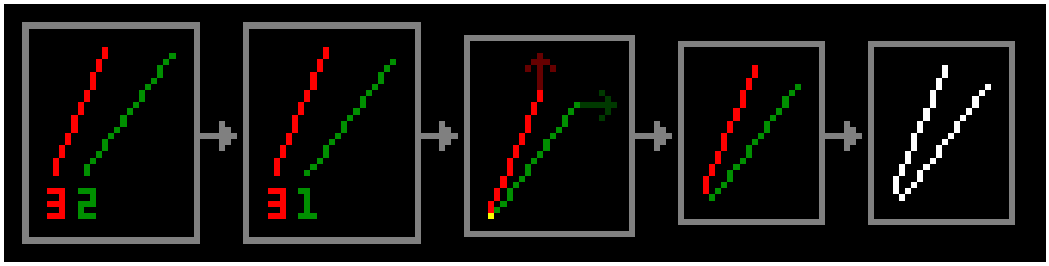


Figure 7.24: To draw an angle with an octant range of 1 with imperfect slopes, apply angle widening and edge offsetting.

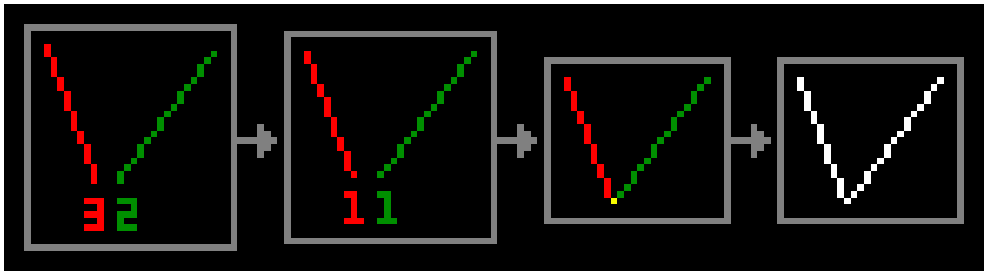
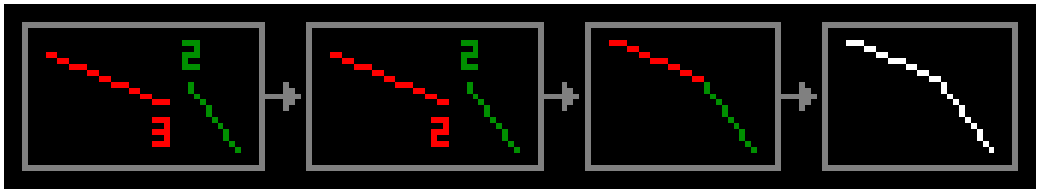


Figure 7.25: To draw an angle with an octant range of 2 or 3 with imperfect slopes, apply angle widening.

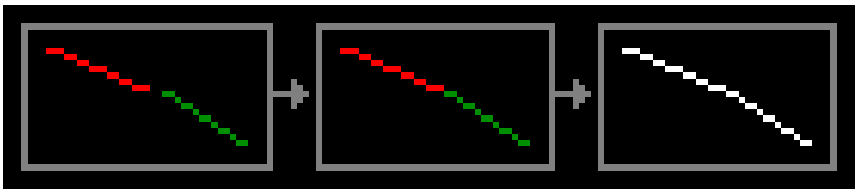
If an angle has an octant range of either 2 or 3, then it simply needs to be widened. Figure 7.25 shows an example of such an angle. In this case, we minimize the length of both initial spans to 1. Then we join them by intersecting the initial pixels to get the resulting angle.

For angles with octant ranges greater than 3 and quadrant ranges of 2, truncate the initial spans and merge them as described in Section 7.4.3. If the quadrant range is 3, then the angle is drawn by joining the edges without intersection, as described in Section 7.4.4.

A complete set of results can be found in Figures F.3, F.4, F.5, and F.6 in Appendix F. The results include angles composed of level-2 and level-3 lines drawn using our algorithm and the naïve method. Although our algorithm draws better angles than the naïve method, the quality of the angles still degrades as the level of the lines increase. Therefore it is always preferable to use low-level lines so that the angles can be drawn more cleanly.



(a)



(b)

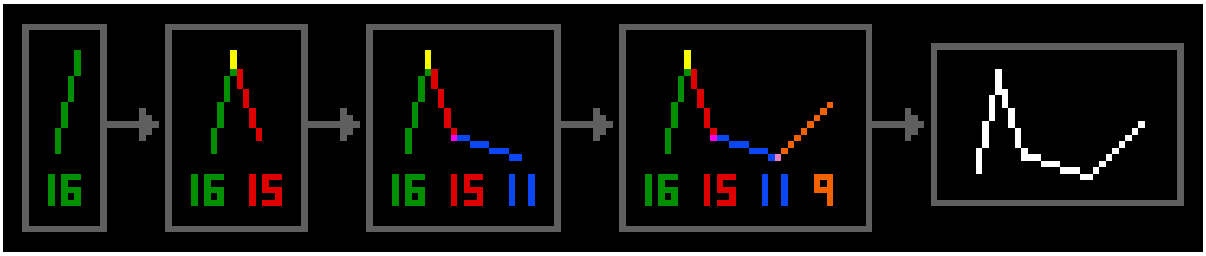
Figure 7.26: For an angle with an octant range greater than 3, (a) if it has a quadrant range of 2, then it is drawn by truncating and merging, and (b) if it has a quadrant range of 3, then it is drawn by joining the edges without overlap.

7.6 Drawing Polygonal Paths

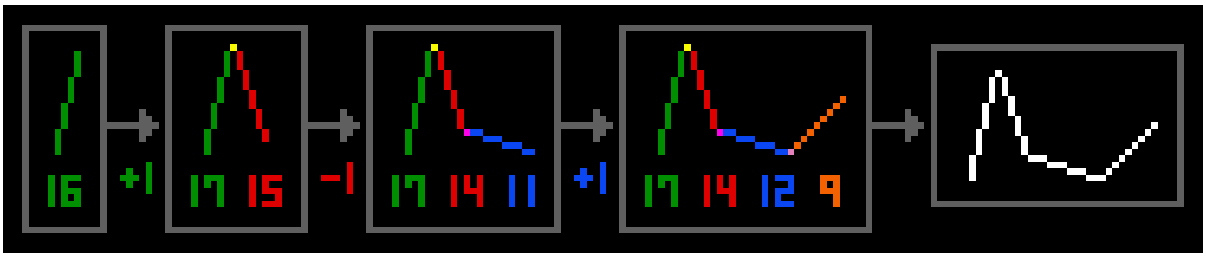
So far, our algorithm draws good angles (i.e., those that follow eBoy's criteria from Section 7.2), but only in isolation. Now we want to draw polygonal paths with good angles. Figure 7.27a shows a naïve method for drawing a polygonal path: every new edge starts where the previous edge ends. The resulting polygonal path has several bad angles. Figure 7.27b shows a better way of drawing a polygonal path: when adding a new edge, modify the length of the previous edge slightly so that a good angle can be drawn. The resulting polygonal path contains only good angles.

Our algorithm for drawing a polygonal path with good angles is simple. We will describe it with the example shown in Figure 7.28. Suppose the first edge in the polygonal path has a slope of $-\frac{1}{5}$. Its initial span length depends on the length of the edge. The top row shows the five possibilities; the number above each one indicates its initial span length. Suppose the second edge has a slope of $\frac{1}{2}$. According to our algorithm, a good angle between these two edges looks like the one shown in the second row.

To draw this good angle, the initial span length needs to be 1. If it is currently not 1, then it needs to be either shortened or lengthened. The third row in Figure 7.28 shows how to compute this length change. Let L_{old} be the old initial span length and L_{new} the new initial span length. Let L_{max} be the maximum span length in the first edge, which is 5



(a)



(b)

Figure 7.27: A polygonal path containing four edges is drawn in two different ways. At each step, a new edge is added and all the edge lengths are labelled. (a) The naïve method draws bad angles but does not change the lengths of any previously drawn edges. (b) Our method draws good angles but may change the length of the previously drawn edge.

in this example. Then we compare the value of $L_{\text{old}} - L_{\text{new}}$ to $\frac{L_{\max}}{2}$. If $L_{\text{old}} - L_{\text{new}} < \frac{L_{\max}}{2}$, then the initial span is shortened. Otherwise it is lengthened. The last row shows the resulting angles as well as the total change in length of the first edge, which is between -2 and 2 inclusive.

With our algorithm, whenever a new edge is added to an old edge, the length of the old length may change by at most $\left\lfloor \frac{L_{\max}}{2} \right\rfloor$. For example, if the old edge has a slope of $-\frac{1}{5}$ (as shown in Figure 7.28), then since its maximum span length is 5 pixels, adding a new edge would change its length by at most $\left\lfloor \frac{5}{2} \right\rfloor = 2$ pixels.

If the maximum span length is long enough, this change in length starts becoming quite noticeable; this is the main drawback of our algorithm. One solution—which our algorithm currently uses—is to place a threshold on the maximum span length. Given an upper bound δ on the edge length change, the maximum span length threshold is $2\delta + 1$. For example, if we want the length of an edge to change by at most 1 pixel when a new edge is added to it, then we can set the threshold for the maximum span length to be $2 \cdot 1 + 1 = 3$. This means, in addition to horizontal and vertical lines, only lines whose

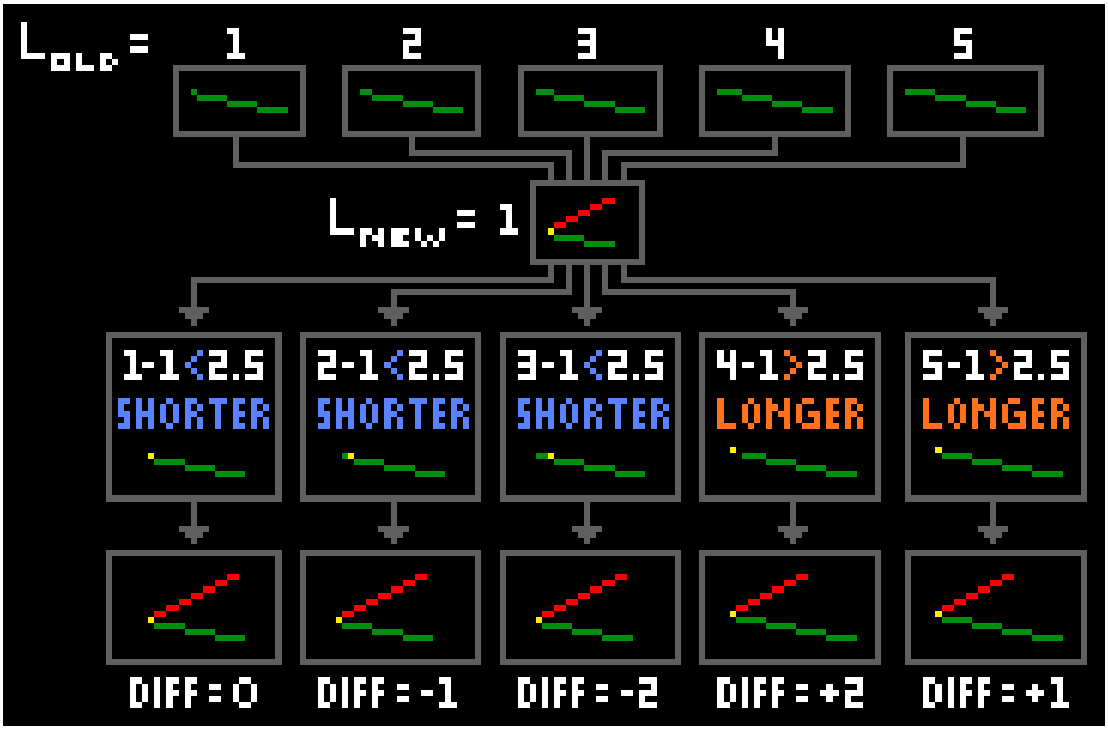
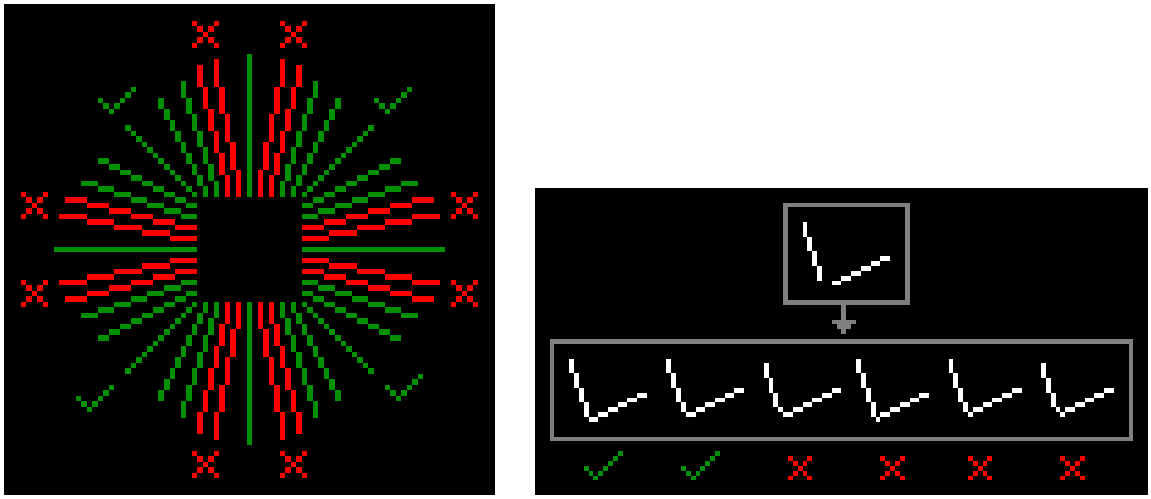


Figure 7.28: How the length of a previously drawn edge may change when drawing a polygonal path.

slopes have absolute values in the range $[\frac{1}{3}, 3]$ are allowed in the drawing. Figure 7.29a shows the resulting set of allowed lines in green and some disallowed lines in red.

Another solution is to consider multiple angle candidates. For example, Figure 7.29b shows six different ways to draw the same angle. If we consider the first two to be satisfactory choices, then when drawing a polygonal path, we can choose whichever candidate minimizes the edge length change. A more complete solution would involve assigning each candidate a score based on how good it looks, and then using this score in conjunction with the edge length change to choose the best candidate. We leave the computation of the score, and its use in an algorithm for drawing polygonal paths, as future work.

Figure 7.30a shows a vector shape (left) containing many different types of angles drawn by our algorithm (middle) and by the naïve method (right). Both pixelated shapes look similar to the vector shape. Our result contains better-looking but sometimes duller angles while the naïve result contains sharper but more artifact-prone angles. Figure 7.30b highlights the pixel clusters in purple to show that our algorithm successfully reduces the



(a) (b)

Figure 7.29: How to limit length changes in previously drawn edges: (a) allow only lines of certain slopes to be drawn, or (b) consider multiple candidates for the angle.

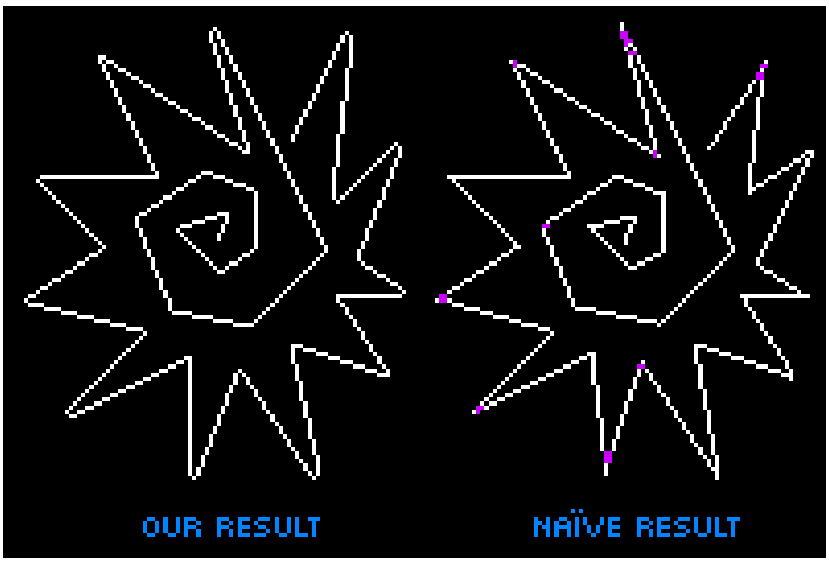
number of such artifacts. Our method, however, does not work for closed loops because it cannot guarantee the final angle to be artifact-free.

7.7 Summary

We studied eBoy's isometric pixel art drawings and conducted a survey with them to determine what makes angles look good at low resolutions. Using this information, we developed an angle-drawing algorithm that effectively characterizes their approach to angle drawing. We also developed an algorithm for drawing polygonal paths one edge at a time; when a new edge is added, the previously drawn edge may be slightly altered so that the two edges form a good angle.

For future work, we would like to improve our polygonal path algorithm. As discussed earlier, we would like to assign quality scores to angles so that multiple angle candidates may be considered when drawing angles in a polygonal path. We also want our algorithm to support more complex structures such as closed loops, multi-way junctions, and angles formed by curves rather than straight lines. Finally, to provide more flexibility, we want to allow artists to manually specify angles that our algorithm can choose from when drawing shapes.

(a)



(b)

Figure 7.30: (a) A polygonal path drawn using our algorithm and a naïve method. (b) The purple pixels indicate pixel clusters.

Chapter 8

Conclusions and Future Work

In this thesis, we presented several pixelation algorithms, a special class of rasterization algorithms inspired by pixel art. Superpixelator suppresses artifacts when pixelating paths and preserves shape-level properties when pixelating shape primitives. In addition, it supports manual antialiasing, which uses a limited palette and produces clean results with minimal blurring. We also developed algorithms specifically for pixelating straight lines and angles formed by them. For these algorithms, the main goal is to reproduce the pixel patterns found in hand-drawn pixel art.

In this chapter, we will provide a summary of our work along with its contributions and limitations. Then we will describe directions for future research, including improvements to our algorithms and other areas of pixel art to explore.

8.1 Superpixelator (without Antialiasing)

In Chapters 3 and 4, we described our pixelation algorithm, Superpixelator, when applied to paths and shapes without antialiasing. When pixelating paths, it uses partial sorting and grid alignment to suppress jaggies and blips. When pixelating shape primitives, Superpixelator considers a set of shape-level properties such as symmetry and aspect ratio, and tries to preserve them via optimization.

Superpixelator was evaluated in two different ways. First we conducted a user study to compare three sets of pixel drawings: (1) vector drawings pixelated by Superpixelator, (2) vector drawings rasterized by commercial software, and (3) hand-drawn pixel art by both amateurs and experts. Our results were rated the highest for both visual appeal and

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

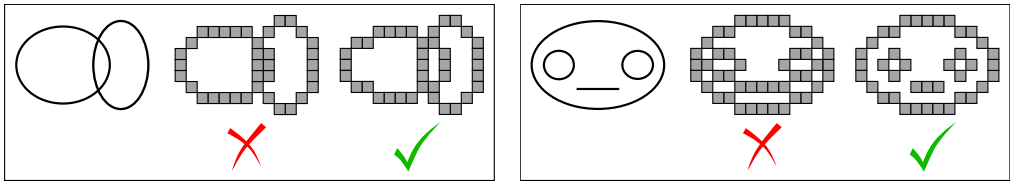
Figure 8.1: Pixel art reproduced by Superpixelator using an interactive drawing tool: (a) Undead Spell Caster by Arachne and (b) its reproduction; (c) Ponyta sprite from Pokémon and (d) its reproduction.

similarity to the vector inputs. For the second stage of the evaluation, we worked with two professional pixel artists to compare Superpixelator's ability to pixelate shape primitives to other rasterizers and pixel artists. The results show that Superpixelator outperforms other rasterizers and is comparable to pixel artists.

Superpixelator performs best in an interactive setting, in which the user can edit the underlying vector shapes directly and see the pixel art drawing update in real time. This way, the user has partial control over pixel-level details without having to painstakingly edit individual pixels. This framework also allows the user to easily apply geometric transformations on shapes and to change the image resolution. Figure 8.1 shows two examples in which we took real pixel art drawings and redrew them using our customized image editor which uses Superpixelator as its rasterizer. Even as amateurs, we were able to draw decent-looking pixel art in a short time (less than ten minutes each). Note that in both cases, the compositions changed slightly since we did not trace them from the original.

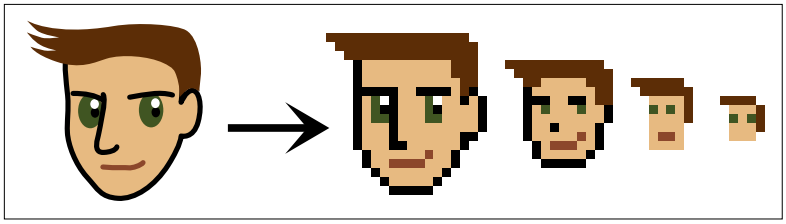
When used in a non-interactive setting, Superpixelator can still draw vector shapes well in isolation, but it can produce artifacts when multiple shapes interact. For example, it cannot always handle multi-way junctions or intersections properly (see Figure 8.2a). Also, pixel-level details may be lost or distorted beyond recognition (see Figure 8.2b). For future work, we would like to explore how to preserve topological properties so that object connections and intersections can be expressed even at low resolutions.

For low-resolution pixel art, we would also like to automate the abstraction process so that a vector image can be rasterized at various resolutions with the appropriate amount

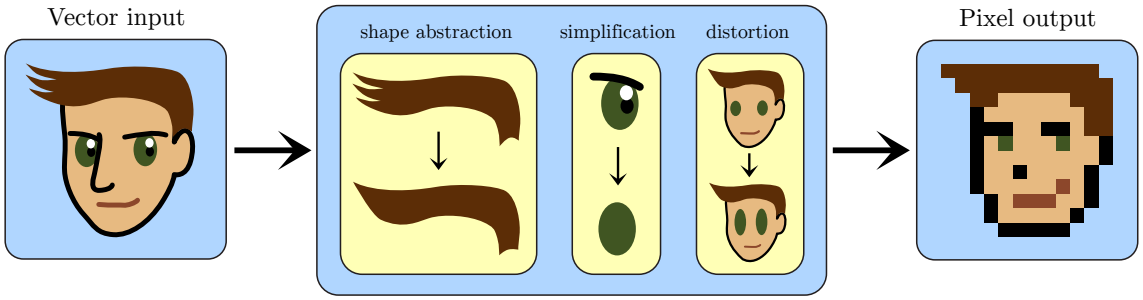


(a) (b)

Figure 8.2: Preserving topology in rasterization includes (a) preserving intersections between shapes, and (b) preserving pixel-level details.



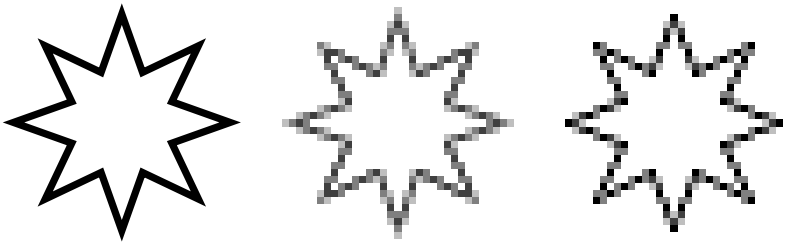
(a)



(b)

Figure 8.3: (a) A vector image pixelated at different resolutions with different levels of abstraction. (b) This effect may be achieved with shape abstraction, simplification, and distortion.

of abstraction and level of detail. Figure 8.3a shows a vector image redrawn as pixel art at various resolutions. Image abstraction involves shape abstraction, simplification and distortion, as shown in Figure 8.3b. We will study how these operations apply to both vector and pixel art.



(a) (b) (c)

Figure 8.4: An 8-pointed star (a) as a vector shape, (b) hand drawn with tapering at the corners, and (c) rasterized by Superpixelator without corner tapering.

8.2 Manual Antialiasing Algorithm

As discussed in Chapter 5, Superpixelator supports a special type of antialiasing called manual antialiasing. To avoid blurriness, we control pixel coverage. We also modify pixel opacity, which changes the apparent thickness, so that the rasterized path appears to be the same thickness as the input vector path. Finally, manual antialiasing uses a limited palette so that it does not introduce too many new colours to the existing palette.

The evaluation process involves comparing our results to those produced by other rasterizers and a pixel artist. In general, our algorithm outperforms other rasterizers and our results bear many similarities to hand-drawn pixel art. However, the pixel artist pointed out that our algorithm sometimes produces irregular pixel patterns when rasterizing straight lines. This issue was addressed in Chapter 6.

One limitation of our algorithm is that it does not handle different joint types, such as mitre joints or rounded joints. Figure 8.4b shows a star drawn by a pixel artist which contains tapering at the corners to indicate mitre joints, compared to one rasterized by Superpixelator in Figure 8.4c which does not have any tapered corners. For future work, we would like to consider how to express different joint types with manual antialiasing. Being able to properly handle corners will allow us to draw filled shapes.

8.3 Euclidean Line-Drawing Algorithm

In Chapter 6, we introduced the Euclidean Line-Drawing Algorithm. It draws straight lines with highly regular pixel patterns both with and without antialiasing. The antialiased

version of the algorithm has several parameters that allow the user to control exactly how much antialiasing to apply and the regularity of the pixel patterns used.

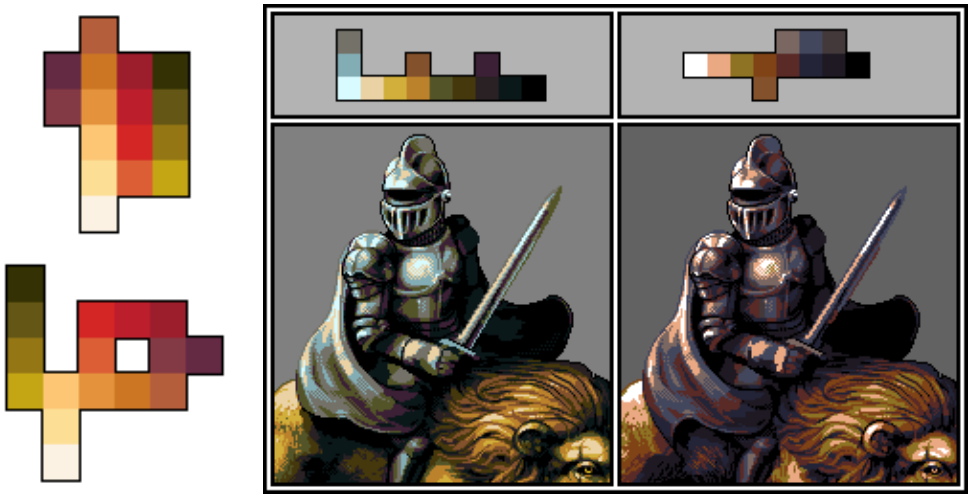
Along with the algorithm, we also introduced two different representations of pixelated lines. If we use the compact representation, then the running time of the algorithm scales with the irregularity of pixel patterns. In other words, it takes the least amount of time to compute the pixel structure of a line that has highly regular pixel patterns. Our algorithm also provides an easy way to measure the quality of a line by its slope, so that we can preferentially snap to lines of higher quality.

Currently the running time of our algorithm scales sublinearly, which appears to be better than Bresenham's linear-time line drawing algorithm. However, due to overhead costs, our algorithm runs faster only when applied to extremely long lines. For future work, we would like to find a more efficient implementation of our algorithm so that it can be used anywhere other line drawing algorithms would currently be used. Also, since most line drawing happens in hardware, we would like to find an efficient hardware solution for our algorithm.

8.4 Angle-Drawing Algorithm

In Chapter 7, we studied how eBoy draws angles in their isometric pixel art compositions, and summarized their method in our angle-drawing algorithm. Compared to the naïve way of drawing angles, our algorithm draws angles that are free of pixel clusters and have much clearer pixel structures. The algorithm has also been extended to draw polygonal paths with these angles. Our results were evaluated by eBoy, and they concluded that our angles are better than those drawn naïvely by other rasterizers and are comparable to their work.

For future work, we would like to improve the way we draw polygonal paths. Currently, adding a new edge may change the length of the previous edge slightly. We want to minimize this change by considering multiple angle candidates. We would also like to extend our algorithm to handle more complex features such as multi-way junctions and closed shapes. Ultimately, our goal is to incorporate the algorithm into a CAD-like tool for drawing isometric pixel art.



(a) (b)

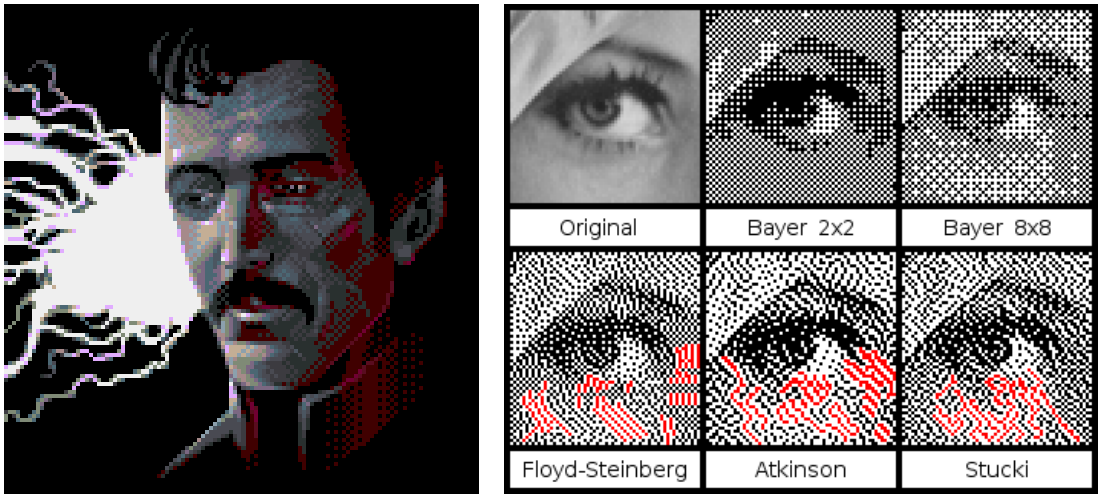
Figure 8.5: (a) Two different ways to arrange the same colour palette by Syosa. (c) Two different palettes applied to the same image by Cure. Used with permission.

8.5 Other Research Opportunities in Pixel Art

In this work, we studied pixel art techniques with the goal of improving rasterization algorithms. However, there are many aspects of pixel art not directly related to rasterization that may provide interesting research opportunities in other areas of computer graphics.

Pixel artists work with limited colour palettes, which presents several challenges. A good palette should contain a small number of colours that work well together. Pixel artists often arrange their palettes in a way that captures the functional relationships between colours, individually or in groups. Figure 8.5a shows two arrangements of the same palette; the top is a simple arrangement of hue and value that is not as informative as the bottom one, which shows how various families of colours interact. A good palette arrangement is not only a good way to visualize a composition, but also makes colour selection and manipulation much easier. The palette has significant effect on the composition, as shown in Figure 8.5b.

Not much research has been done in the way of limited palette arrangements. Most image editors use fairly primitive palette visualizations. For example, GIMP's colourmap permits either manual rearrangement, or HSV sorting. However, there is some related work on the visualization, categorization, and interaction of colours. Haber et al. [25] proposed a



(a) (b)

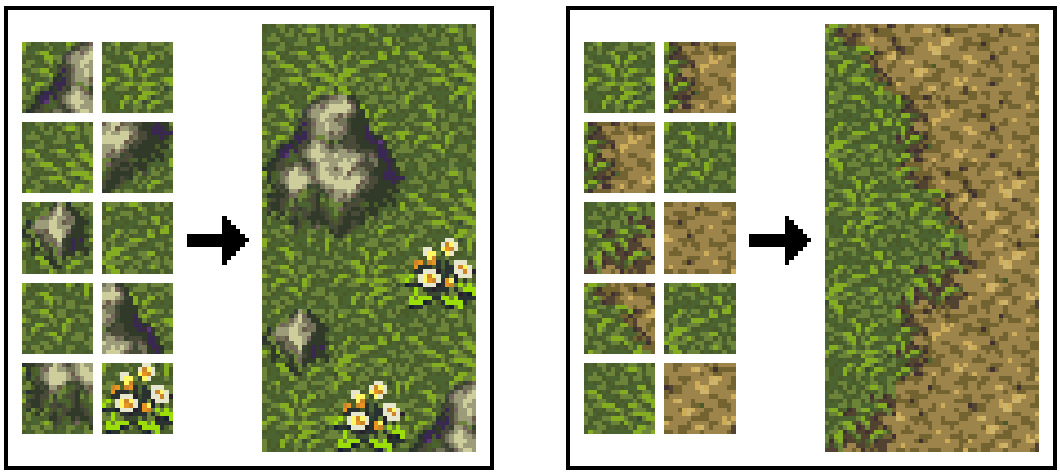
Figure 8.6: (a) *Nikola Tesla, Man of Science* by Helm shows how pixel artists apply dithering. (b) A comparison of dithering algorithms. Pixel strands are highlighted in red. Used with permission.

histogram visualization for organizing colours in paintings. Heer and Stone [28] developed a colour model based on the XKCD colour-naming data set. The Paper app for iPad contains a data-driven colour-mixing tool that mixes colours in an intuitive way [14]. Meier et al. [47] created interactive colour palette tools to aid users in selecting and experimenting with colours within the context of their compositions.

When using limited colours, dithering helps create smoother colour transitions. Pixel artists typically use ordered dithering to create gradients. Sometimes stylized dithering is used to suggest texture. Figure 8.6a shows an example of how pixel artists use dithering in their artwork.[1] Although there are many dithering algorithms, they are rarely used in pixel art because they produce artifacts such as long strands of pixels (as shown in red in Figure 8.6b). Although pixel artists use ordered dithering, their results differ from that of a typical automated algorithm. The top row of Figure 8.6b shows two different types of ordered dithering, with 2×2 and 8×8 Bayer matrices. Studying how pixel artists apply dithering can help improve dithering algorithms in the context of pixel art.

In pixel art games, tile sets are often used to create the background. By designing texture tiles that fit together, they can be used efficiently to create more variety in the

[1] http://fiji.sc/wiki/index.php/Dithering



(a) (b)

Figure 8.7: Two tilesets from Secret of Mana (1993).

background. Figure 8.7 shows two tilesets from a game, used to create two different backgrounds. The process of designing tile sets can be tedious, but we may be able to apply texture synthesis techniques such as image quilting to simplify the process [13, 16].

We believe that studying various aspects of pixel art can lead to new and inventive ways of solving related problems in computer graphics. As we have shown in our work, analyzing techniques developed by pixel artists can provide new perspectives on topics as fundamental as rasterization. Studying pixel art may also lead to new font rasterization techniques that reduce the need for extensive font hinting. Being able to create pixel art more efficiently also provides artists with more creative opportunities, such as making feature-length pixel art animations. We hope to continue to improve our algorithms so that they can be easily incorporated into the standard pixel art workflow and that our work will inspire more research on the topic of pixel art.

APPENDICES

Appendix A

Partial Sorting Parameters

In Section 3.3.4, we described a partial sorting algorithm that involves defining a cost for pixelated paths in terms of positional deviation D_p, slope deviation D_s, and sortedness S. Here we provide details on how to compute each quantity, including the pseudocode. Note that we are assuming the input `vectorPath` is not a straight line, since straight lines do not need to be sorted.

Positional deviation D_p is calculated by finding the shortest distance between the vector path and the centre of each pixel in the pixelated path, and then take the maximum of these values.

```
getPositionalDeviation(vectorPath, pixelatedPath) {
   deviations = []
   for (pixel in pixelatedPath) {
      pc = pixel.getCentre()
      dev = vectorPath.getShortestDistanceTo(pc)
      deviations.add(dev)
   }
   return deviations.getMaximum()
}
```

For slope deviation D_s, we know that each pixel span corresponds to a line segment that approximates a section of the vector curve. So we compare them by taking angle difference between their slopes; the slope of the pixel span is the slope of its diagonal, and its corresponding slope on the vector curve is the tangent slope at the point closest to the pixel span's centre. D_s is defined as the maximum of these angle differences.

```
getSlopeDeviation(vectorPath, pixelatedPath) {
   deviations = []
   for(pixelSpan in pixelatedPath) {
      slope1 = pixelSpan.getDiagonalSlope()
      spanCentre = pixelSpan.getCentre()
      closestVectorPoint = vectorPath.getPointClosestTo(spanCentre)
      slope2 = vectorPath.getSlopeAt(closestVectorPoint)
      angleDiff = getAngleDifference(slope1, slope2)
      deviations.add(angleDiff)
   }
   return deviations.getMaximum()
}
```

Sortedness S how much of a pixelated path is sorted order. Suppose a pixelated path corresponding to the vector path with positive curvature has the slope sequence $\{s_0, s_1, \ldots, s_n\}$. Then its sortedness is defined as the number of pairs $\{s_i, s_j\}$ where $i < j$ that satisfy $s_i \leq s_j$.

```
getSortedness(vectorPath, pixelatedPath) {
   slopeSeq = pixelatedPath.getSlopeSequence()
   n = slopeSeq.getLength()
   curvature = vectorPath.getCurvature()
   numSorted = 0
   for i from 0 to n {
      for j from i to n {
         if(curvature > 0 && slopeSeq[i] <= slopeSeq[j]) numSorted++
         else if (curvature < 0 && slopeSeq[i] >= slopeSeq[j]) numSorted++
      }
   }
   return numSorted
}
```

Appendix B

User Study Statistical Analysis

We provide a derivation for the statistic used for hypothesis testing in the user study described in Chapter 3. Let X_{ij} denote the j-th response of participant i. We know that X_{ij} are independent and identically distributed with $X_{ij} \sim \text{Bernoulli}(p_i)$ for $i = 1, 2, \ldots, m_i$, where the mean is $\mu_i = p_i$ and the variance is $\sigma_i^2 = p_i(1 - p_i)$. Let

$$S_i = \frac{1}{m_i} \sum_{j=1}^{m_i} X_{ij} \tag{B.1}$$

Then by the Central Limit Theorem, as n approaches infinity,

$$\sqrt{m_i}(S_i - \mu_i) \xrightarrow{d} \mathcal{N}\left(0, \sigma_i^2\right) \tag{B.2}$$

$$\Rightarrow S_i - \mu_i \xrightarrow{d} \mathcal{N}\left(0, \frac{\sigma_i^2}{m_i}\right) \tag{B.3}$$

$$\Rightarrow S_i \xrightarrow{d} \mathcal{N}\left(\mu_i, \frac{\sigma_i^2}{m_i}\right). \tag{B.4}$$

Next, let

$$S = \frac{1}{n} \sum_{j=1}^{n} S_i. \tag{B.5}$$

Since the sum of two independent normal variables is also normally distributed, we have

$$\sum_{i=1}^{n} S_i \xrightarrow{d} \mathcal{N}\left(\sum_{i=1}^{n} \mu_i, \sum_{i=1}^{n} \frac{\sigma_i^2}{m_i})\right) \tag{B.6}$$

$$\Rightarrow S \xrightarrow{d} \mathcal{N}\left(\frac{1}{n}\sum_{i=1}^{n} \mu_i, \frac{1}{n^2}\sum_{i=1}^{n} \frac{\sigma_i^2}{m_i}\right) \tag{B.7}$$

We can normalize this to

$$\frac{S - \frac{1}{n}\sum_{i=1}^{n} \mu_i}{\frac{1}{n}\sqrt{\sum_{i=1}^{n} \frac{\sigma_i^2}{m_i}}} \xrightarrow{d} \mathcal{N}(0,1). \tag{B.8}$$

We want to see if the participant significantly favours the first group over the second group. The null hypothesis is

$$H_0: \quad \frac{1}{n}\sum_{i=1}^{n} \mu_i \leq 0.5. \tag{B.9}$$

Calculate the z-value

$$\frac{\hat{S} - 0.5}{\frac{1}{n}\sqrt{\sum_{i=1}^{n} \frac{\hat{\sigma}_i^2}{m_i}}} \xrightarrow{d} \mathcal{N}(0,1). \tag{B.10}$$

where

$$\hat{S} = \frac{1}{n}\sum_{i=1}^{n} \hat{p}_i, \tag{B.11}$$

$$\hat{\sigma}_i^2 = \hat{p}_i(1 - \hat{p}_i), \tag{B.12}$$

$$\hat{p}_i = \frac{1}{m_i}\sum_{j=1}^{m_i} x_{ij}. \tag{B.13}$$

Then find the p-value by taking the Normal CDF of the z-value. If the p-value is less than 0.05, reject the null hypothesis.

Appendix C

User Study Questionnaire

The following questionnaire is given to participants of the user study described in Section 3.4.

1. Age Range:
   - ☐ 18 to 27 years
   - ☐ 28 to 37 years
   - ☐ 38 to 47 years
   - ☐ 48 to 57 years
   - ☐ 58 to 67 years
   - ☐ 68 years and over
2. Gender:
   - ☐ Male
   - ☐ Female
3. What type(s) of art have you done in the past? Check all that apply.
   - ☐ Painting
   - ☐ Sketching
   - ☐ Sculpting

- ☐ Graphics design
- ☐ Creating comics
- ☐ Image processing
- ☐ Type design/calligraphy
- ☐ Filmmaking
- ☐ Other; please specify _____

4. How familiar are you with pixel art?

   - ☐ Not at all (skip to Question 6)
   - ☐ I have seen it before (skip to Question 6)
   - ☐ I know how it is created (skip to Question 6)
   - ☐ I have created pixel art before (go to Question 5)

5. If you have created pixel art, how long have you been practicing it?

   - ☐ Less than a month
   - ☐ One to six months
   - ☐ Six months to a year
   - ☐ One to five years
   - ☐ Five to ten years
   - ☐ More than ten years

6. Please provide any additional information including how you heard about this study, where we can find samples of your pixel art works, and anything you would like to share.

7. If you would like to receive updates or participate in Part 2 of this study, please provide your e-mail address (we promise not to spam!).

Appendix D

User Study Demographic Information

The demographic information collected from the 148 participants of the Drawing Phase of user study (see Section 3.4) are summarized in the following bar plots. For some of the plots, the sum of the values may be less than the total number of participants due to non-responses. Note that artistic level refers to the number of checked items in Question 3 of the questionnaire (see Appendix C).

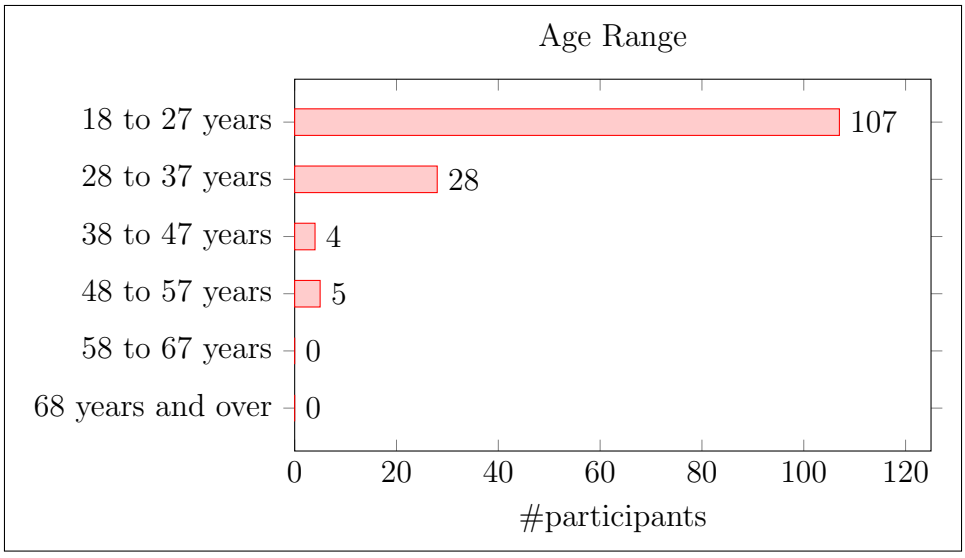


Figure D.1: Age range of user study participants

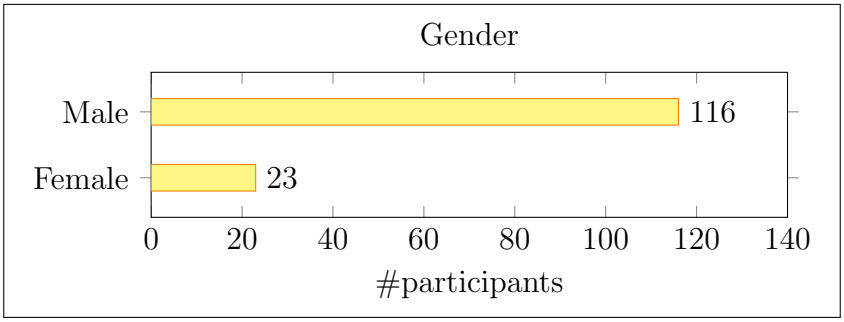


Figure D.2: Gender distribution of user study participants

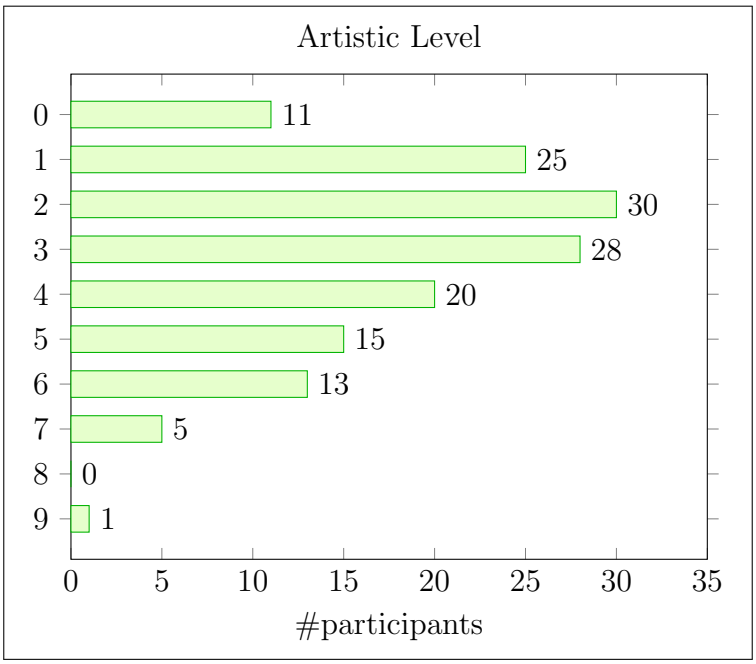


Figure D.3: Artistic level among user study participants

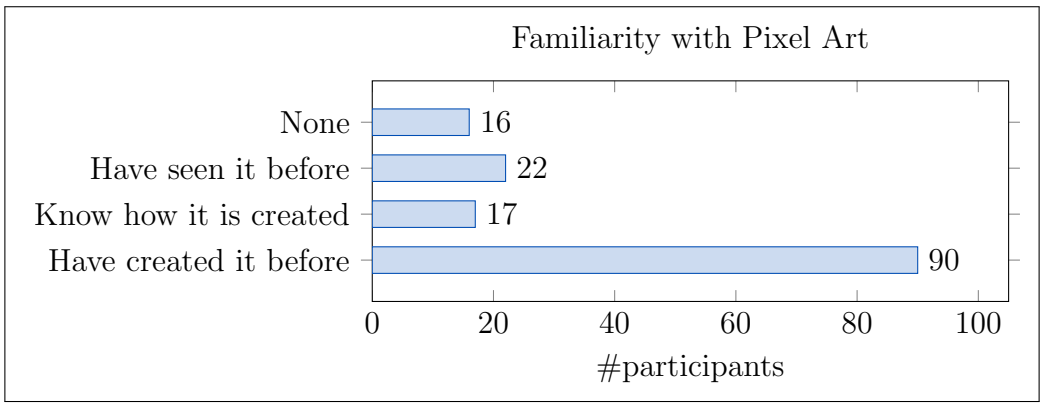


Figure D.4: Familiarity with pixel art among user study participants

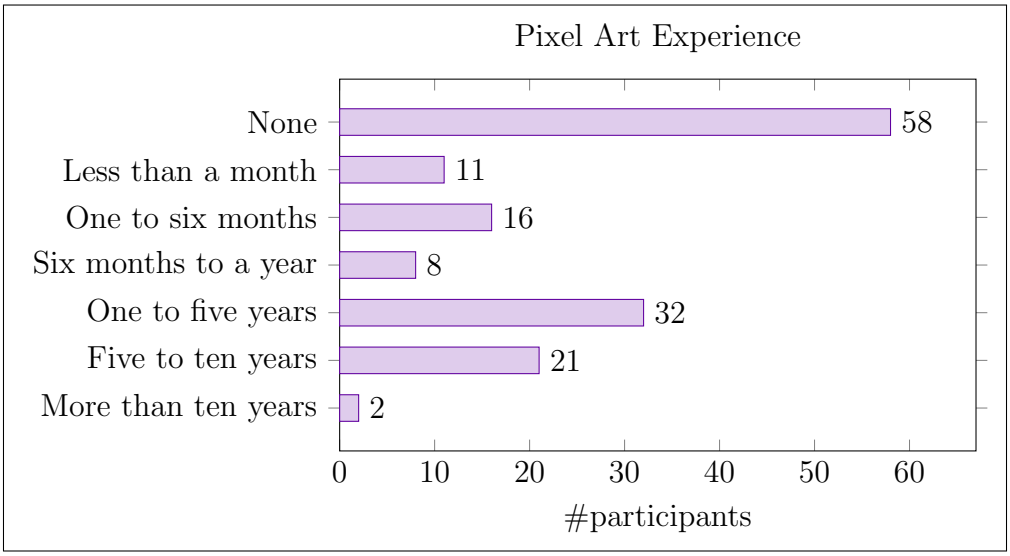


Figure D.5: Pixel art experience of user study participants

Appendix E

ELDA-A Error Bound

In Chapter 6, we described ELDA-A, a recursive line-drawing algorithm based on the Euclidean algorithm. We would like to determine an error bound for the result of this algorithm. First, define the *error* between two paths as the maximum perpendicular distance between them. More precisely, if the two paths are parametrized by $P(t)$ and $Q(t)$, then the error is given by

$$E_{P,Q} = \max_{0 \le t \le 1} d_P(Q(t)) \tag{E.1}$$

$$= \max_{0 \le t \le 1} d_Q(P(t)), \tag{E.2}$$

where $d_P(Q(t))$ is perpendicular (i.e., shortest) distance from the point $Q(t)$ to the path P, and $d_Q(P(t))$ is defined similarly.

To calculate the error of our algorithm, we calculate the error between the input vector line and polygonal path representing the pixelated output. For example, Figure E.1a shows a vector line of slope 4/3 and Figure E.1b shows the pixelated approximation of it, with its polygonal representation shown in blue. Then the error is the length of the black segment in Figure E.1c.

We want to find an upper bound for the error between a given vector line and its pixelation by ELDA-A. Denote the error by E. For the rest of this derivation, since we are going to be referring to polygonal path representations of pixelated lines rather than the actual pixels, $\mathbf{c}_i$ will refer to the vector from the first pixel corner to the second pixel corner. To find an error bound, first observe that our algorithm is recursive, and at every step we are making an approximation. The sum of all these errors must add up to a value greater than the error between the input and the output. Figure E.2 shows how intermediate

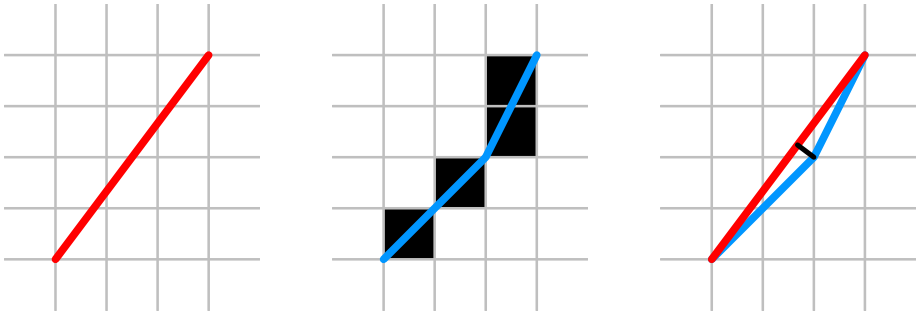


Figure E.1: (a) A vector line joining two pixel corners, (b) its pixelated approximation represented as a polygonal path, and (c) the error calculated as the maximum perpendicular distance between the two paths.

segments are constructed in ELDA-A. At Step i, we are approximating $\mathbf{c}_i$ (shown in red) by a combination of $\mathbf{c}_{i-2}$ and $\mathbf{c}_{i-1}$ (shown in blue). Let E_i be the error between these two paths. Then we have the following inequality:

$$E \leq \sum_{i=0}^{\ell} E_i, \tag{E.3}$$

where ℓ is the number of steps in the algorithm. For the remaining derivation, we will assume without loss of generality that the line we want to draw has slope $m \geq 1$, so that we can conveniently refer back to an example used in Chapter 6 in which $m = \frac{28}{22}$.

Next we want to find an upper bound for each E_i. Consider Lemma 1:

Lemma 1. *Let E_i be the error introduced at Step i of ELDA-A (see Figure E.3a). Let $\bar{\bar{E}}_i$ be the distance of the projection of $\mathbf{c}_i$ onto $\mathbf{c}_{i-1}$, if it were extended infinitely (see Figure E.3b). Then we have*

$$E_i \leq \bar{\bar{E}}_i = \frac{\Delta x_i \Delta y_i}{\sqrt{\Delta x_i^2 + \Delta y_i^2}}, \tag{E.4}$$

Proof. First we must define E_i. Figure E.3a shows that, at Step i, the algorithm approximates $\mathbf{c}_i$ with q_i copies of $\mathbf{c}_{i-1}$ and one $\mathbf{c}_{i-2}$, and E_i is basically the height of this triangle. Let us first get some upper bounds on E_i. Take the perpendicular distance from $\mathbf{c}_i$ to $q_i\mathbf{c}_{i-1}$ and call this value $\bar{E}_i$ (see Figure E.3b). Since E_i is the height of the triangle, it

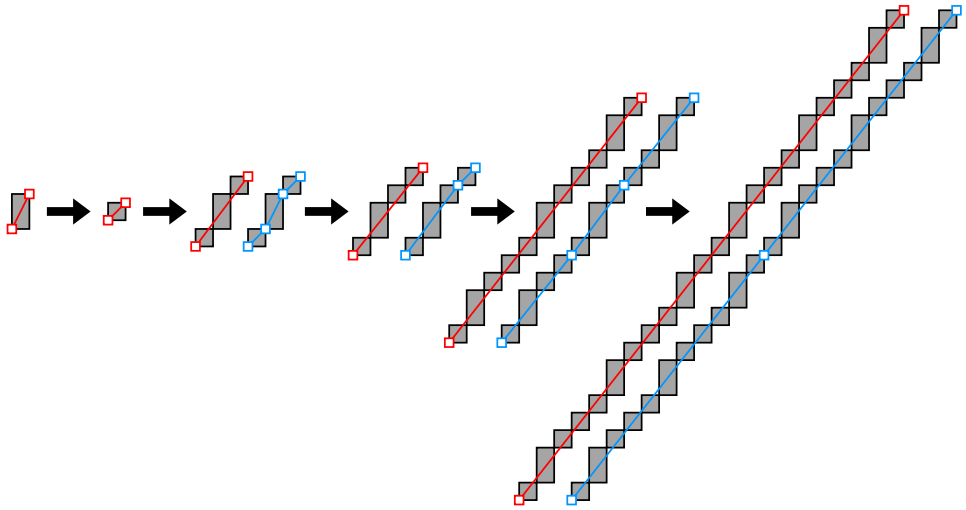


Figure E.2: The error introduced at each step of the algorithm.

must be that $E_i \leq \bar{E}_i$. Taking it one step further, if we extend $\mathbf{c}_{i-1}$ further as shown in Figure E.3c, then take the perpendicular distance from $\mathbf{c}_i$ to it, then this distance—call it $\bar{\bar{E}}_i$—must be greater than or equal to $\bar{E}_i$, by similar triangles. Thus we have

$$E_i \;\leq\; \bar{E}_i \;\leq\; \bar{\bar{E}}_i \;=\; \frac{\Delta x_i \Delta y_i}{\sqrt{\Delta x_i^2 + \Delta y_i^2}}, \tag{E.5}$$

where Δx_i and Δy_i are respectively the horizontal and vertical distances between the extended line and the endpoint of $\mathbf{c}_i$, as shown in Figure E.3c.

□

Lemma 1 gives us a new error bound:

$$E \;\leq\; \sum_{i=0}^{\ell} E_i \;\leq\; \sum_{i=0}^{\ell} \frac{\Delta x_i \Delta y_i}{\sqrt{\Delta x_i^2 + \Delta y_i^2}}. \tag{E.6}$$

Now let us write this expression in terms of x_i and y_i, the width and height of $\mathbf{c}_i$ respectively. Table E.1 shows values from an example in Chapter 6. The way x_i and y_i are computed recursively in our algorithm, they satisfy Lemma 2:

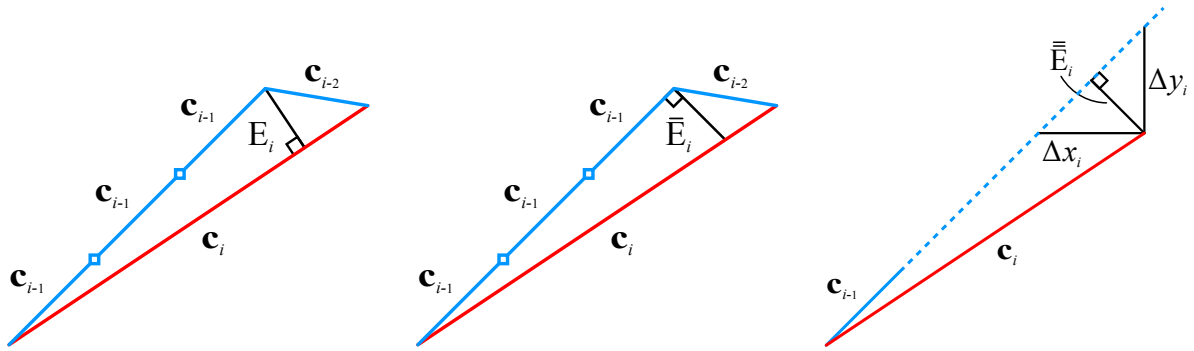


Figure E.3: (a) E_i is the height of the triangle. (b) $\bar{E}_i$ is the perpendicular distance from $\mathbf{c}_i$ to $q_i\mathbf{c}_{i-1}$. (c) $\bar{\bar{E}}_i$ is the perpendicular distance from an extension of $\mathbf{c}_{i-1}$ to $\mathbf{c}_i$. This construction gives us the inequality $E_i \leq \bar{E}_i \leq \bar{\bar{E}}_i$.

Lemma 2. *Let* $\{x_i\}$ *and* $\{y_i\}$ *be two sequence defined recursively as follows:*

$$x_i = \begin{cases} 1 & \text{if } i = -2 \\ 1 & \text{if } i = -1 \\ x_{i-2} + q_i x_{i-1} & \text{if } i \geq 0, \end{cases} \tag{E.7}$$

$$y_i = \begin{cases} n+1 & \text{if } i = -2 \\ n & \text{if } i = -1 \\ y_{i-2} + q_i y_{i-1} & \text{if } i \geq 0 \end{cases} . \tag{E.8}$$

i	a_i	b_i	q_i	x_i	y_i
-2				1	2
-1				1	1
0	16	6	2	3	4
1	6	4	1	4	5
2	4	2	2	11	14
3	2	0		22	28

Table E.1: Variables in each step of the recursion in ELDA-A. At Step i, a_i, b_i, q_i and r_i are values computed by the Euclidean algorithm. x_i and y_i are respectively the width and height of $\mathbf{c}_i$, the intermediate segment constructed at Step i.

Then for all $i \geq -2$,

$$x_i y_{i+1} - x_{i+1} y_i \;=\; (-1)^i. \tag{E.9}$$

Proof. We will prove the lemma by induction. These initial values satisfies the base case:

$$|x_{-2} y_{-1} - x_{-1} y_{-2}| \;=\; 1 \cdot (n+1) - 1 \cdot n \;=\; 1 \;=\; (-1)^{-2}. \tag{E.10}$$

Now assuming the lemma holds for all $i \leq k$ for some integer $k \geq -2$, then

$$\begin{aligned}
x_{k+1} y_{k+2} - x_{k+2} y_{k+1} &= x_{k+1}(y_k + q_{k+2} y_{k+1}) - (x_k + q_{k+2} x_{k+1}) y_{k+1} && \text{(E.11)} \\
&= (x_{k+1} y_k - x_k y_{k+1}) + q_{k+2}(x_{k+1} y_{k+1} - x_{k+1} y_{k+1}) && \text{(E.12)} \\
&= -(x_k y_{k+1} - x_{k+1} y_k) && \text{(E.13)} \\
&= -(-1)^k && \text{(E.14)} \\
&= (-1)^{k+1}. && \text{(E.15)}
\end{aligned}$$

Since the lemma also holds for $i = k+1$, we have proven that it is true for all $i \geq -1$. □

Lemma 2 leads to a simple corollary:

Corollary 1. *Let $\{x_i\}$ and $\{y_i\}$ be defined as in Lemma 2. Then for all $i \geq -2$,*

$$|x_i y_{i+1} - x_{i+1} y_i| \;=\; 1. \tag{E.16}$$

Now we can make the following simplification:

$$\Delta x_i \;=\; \left| x_i - \frac{x_{i-1}}{y_{i-1}} y_i \right| \;=\; \frac{x_i y_{i-1} - x_{i-1} y_i}{y_{i-1}} \;=\; \frac{1}{y_{i-1}}. \tag{E.17}$$

Similarly,

$$\Delta y_i \;=\; \frac{1}{x_{i-1}}. \tag{E.18}$$

By combining the two,

$$E \;\leq\; \sum_{i=0}^{\ell} \frac{\Delta x_i \Delta y_i}{\sqrt{\Delta x_i^2 + \Delta y_i^2}} \;\leq\; \sum_{i=0}^{\ell} \frac{\frac{1}{y_{i-1}} \frac{1}{x_{i-1}}}{\sqrt{\frac{1}{y_{i-1}^2} + \frac{1}{x_{i-1}^2}}} \;\leq\; \sum_{i=0}^{\ell} \frac{1}{\sqrt{x_{i-1}^2 + y_{i-1}^2}}. \tag{E.19}$$

i	-2	-1	0	1	2	3	4	5
F_i	1	1	2	3	5	8	13	21
G_i	$n+1$	n	$2n+1$	$3n+1$	$5n+2$	$8n+3$	$13n+5$	$21n+8$

Table E.2: $\{F_i\}$ is the Fibonacci sequence, and $\{G_i\}$ is a similar sequence with the same recursive relation that starts with $n+1$ and n.

To simplify the expression further, notice that since $q_i \geq 1$ for all i,

$$x_i \;=\; \begin{cases} 1 & \text{if } i = -2 \\ 1 & \text{if } i = -1 \\ x_{i-2} + q_i x_{i-1} & \text{if } i \geq 0. \end{cases} \;\geq\; \begin{cases} 1 & \text{if } i = -2 \\ 1 & \text{if } i = -1 \\ x_{i-2} + x_{i-1} & \text{if } i \geq 0, \end{cases} \;=\; F_i, \tag{E.20}$$

where $\{F_i\}$ is the Fibonacci sequence defined by $F_{-2} = F_{-1} = 1$ and $F_i = F_{i-2} + F_{i-1}$ for $i \geq 0$, and

$$y_i \;=\; \begin{cases} n+1 & \text{if } i = -2 \\ n & \text{if } i = -1 \\ y_{i-2} + q_i y_{i-1} & \text{if } i \geq 0 \end{cases} \;\geq\; \begin{cases} n+1 & \text{if } i = -2 \\ n & \text{if } i = -1 \\ y_{i-2} + y_{i-1} & \text{if } i \geq 0 \end{cases} \;\geq\; G_i, \tag{E.21}$$

where $\{G_i\}$ is another Fibonacci-like sequence defined by $G_{-2} = n+1$, $G_{-1} = n$ and $G_i = G_{i-2} + G_{i-1}$ for $i \geq 0$. When $n = 1$, it is known as the Lucas series. The first few terms in $\{F_i\}$ and $\{G_i\}$ are shown in Table E.2.

Let $H_i = \sqrt{F_i^2 + G_i^2}$. Then our earlier error bound can be written as

$$E \;\leq\; \sum_{i=-1}^{\ell-1} \frac{1}{H_i} \;\leq\; \sum_{i=-1}^{\infty} \frac{1}{H_i}. \tag{E.22}$$

Summing to infinity will be easier to calculate later. We want to reduce this expression to a numerical value. To do so, first notice that since sequence $\{G_i\}$ starts with $n+1$ and n but recurses in the same way as $\{F_i\}$, it must be true that $G_i \geq nF_i$, which implies

$$\frac{1}{H_i} \;\leq\; \frac{1}{\sqrt{F_i^2 + (nF_i)^2}} \;\leq\; \frac{1}{\sqrt{1+n^2}F_i}. \tag{E.23}$$

Therefore, we have

$$E \;\leq\; \sum_{i=-1}^{\infty} \frac{1}{H_i} \;\leq\; \sum_{i=-1}^{k} \frac{1}{H_i} + \frac{1}{\sqrt{1+n^2}} \sum_{i=k+1}^{\infty} \frac{1}{F_i}. \tag{E.24}$$

n	1	2	3	4	5	6	7
Error bound	1.4932	0.9681	0.6987	0.5431	0.4432	0.3739	0.3232

Table E.3: Error bound values of a pixelated line for different values of n, where n is the shortest span length in the line.

Since the bound $\frac{1}{H_i} \leq \frac{1}{\sqrt{1+n^2}F_i}$ is not very tight, we can compute the first few terms in the summation, and replace the remainder with the upper bound. The sum of the reciprocals of the Fibonacci numbers is a known irrational number without a closed form:

$$\psi \;=\; \sum_{i=-2}^{\infty} \frac{1}{F_i} \;\approx\; 3.359885666243\ldots. \tag{E.25}$$

So the error bound becomes

$$E \;\leq\; \sum_{i=-1}^{k} \frac{1}{H_i} + \frac{1}{\sqrt{1+n^2}} \sum_{i=k+1}^{\infty} \frac{1}{F_i} \tag{E.26}$$

$$\leq\; \sum_{i=-1}^{k} \frac{1}{H_i} + \frac{1}{\sqrt{1+n^2}} \left(\psi - \sum_{i=-2}^{k} \frac{1}{F_i} \right). \tag{E.27}$$

The error bounds for different values of n are given in Table E.3. These values suggest that, as the lengths of the spans increase, the error decreases. Unfortunately, we did not obtain a tight error bound, and for small values of n, the values of the error bounds are rather large. We believe the problem stems from using the inequality $E \leq \sum E_i$; we experimented with several examples and even when the pixelated line appears to be a good approximation of the vector input, the value of $\sum E_i$ can still be quite large.

Appendix F

Angle-Drawing Algorithm Results

In Chapter 7, we described an angle-drawing algorithm for angles formed by two straight edges. In Section 7.5.1, the notion of *levels* is defined for lines with different slopes. Here our results are organized by levels. Specifically, we have three sets of angles, formed by edges of level 1, 2, and 3. Each set is then subject to two algorithms: (1) our angle-drawing algorithm and (2) a naïve method that involves drawing the edges separately and then joining them at their initial pixels.

Each set of results is arranged as a matrix of angles. Within each column, edge 1 stays fixed. From left to right, edge 1 changes in slope, sweeping the upper half-plane in a counterclockwise fashion. As we go down each column, edge 2 rotates in the counterclockwise direction and the angle widens until it becomes a straight angle. The rate at which the angles widen varies among the columns.

First we have the angles formed by level 1 edges. Figure F.1 shows our results and Figure F.2 shows the naïve results. Next we have the angles formed by level 2 edges. Figure F.3 shows our results and Figure F.4 shows the naïve results. Finally we have the angles formed by level 3 edges. Figure F.5 shows our results and Figure F.6 shows the naïve results.

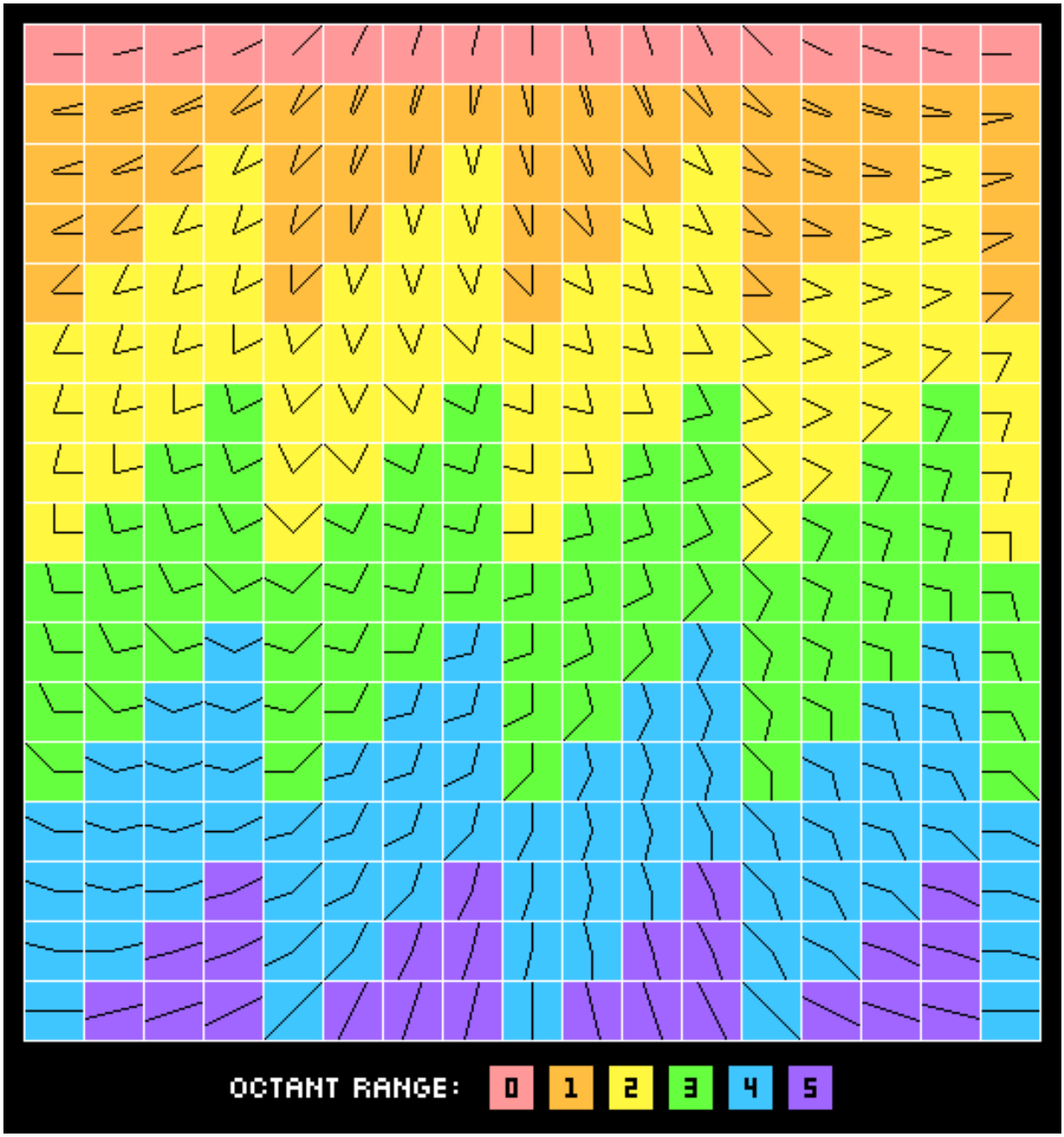


Figure F.1: Angles composed of level 1 edges drawn with our algorithm

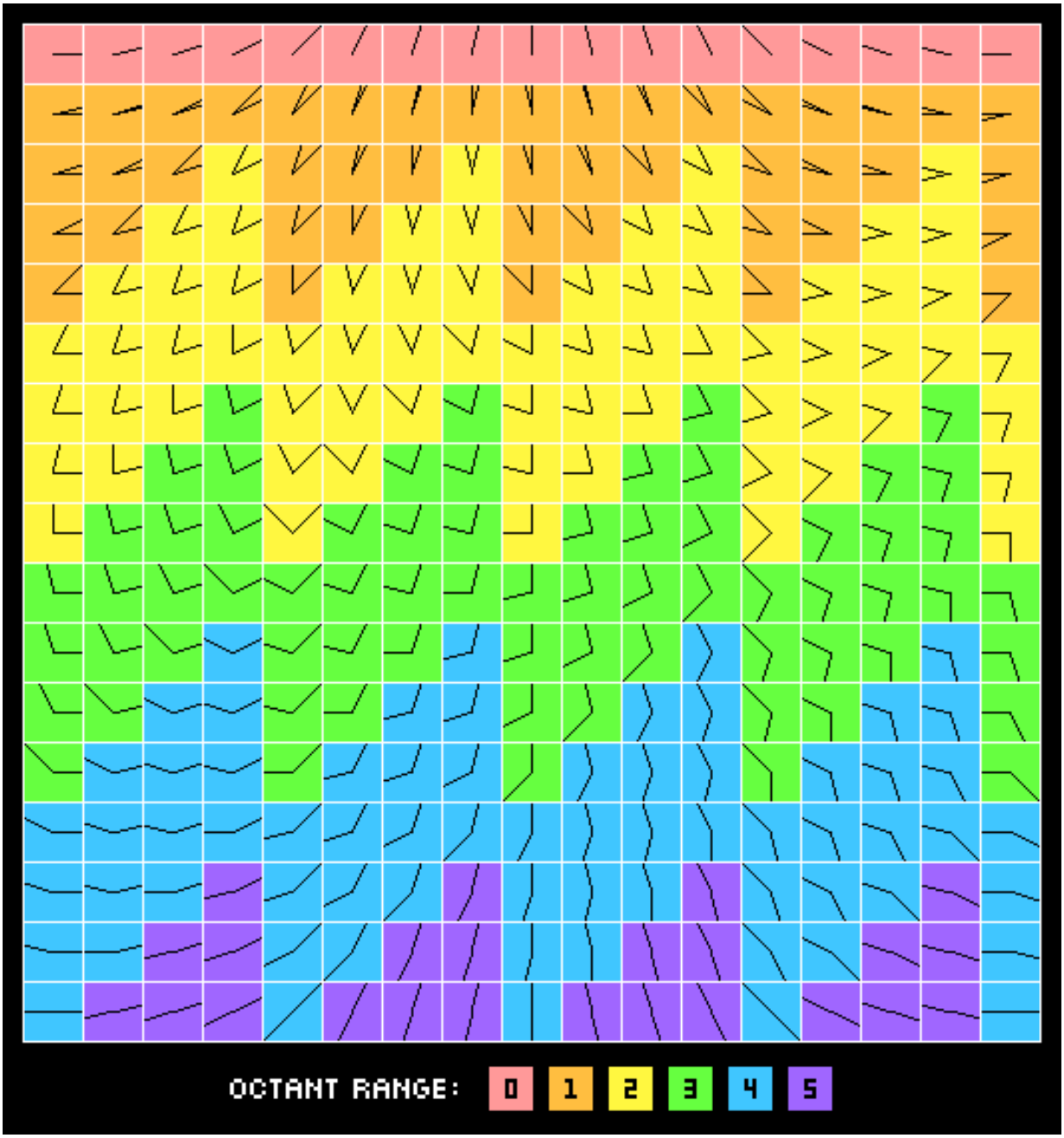


Figure F.2: Angles composed of level 1 edges drawn with the naïve algorithm

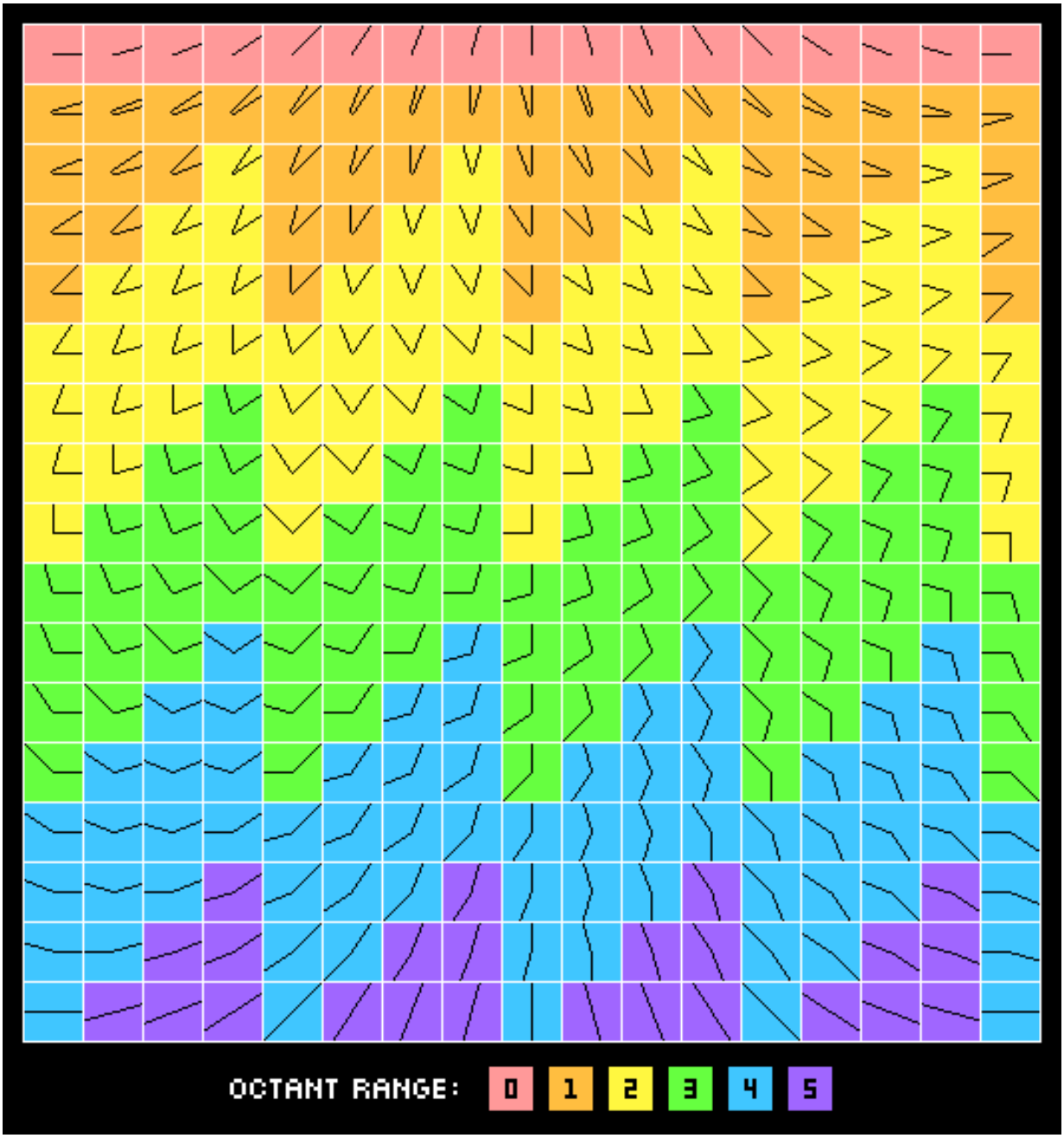


Figure F.3: Angles composed of level 2 edges drawn with our algorithm

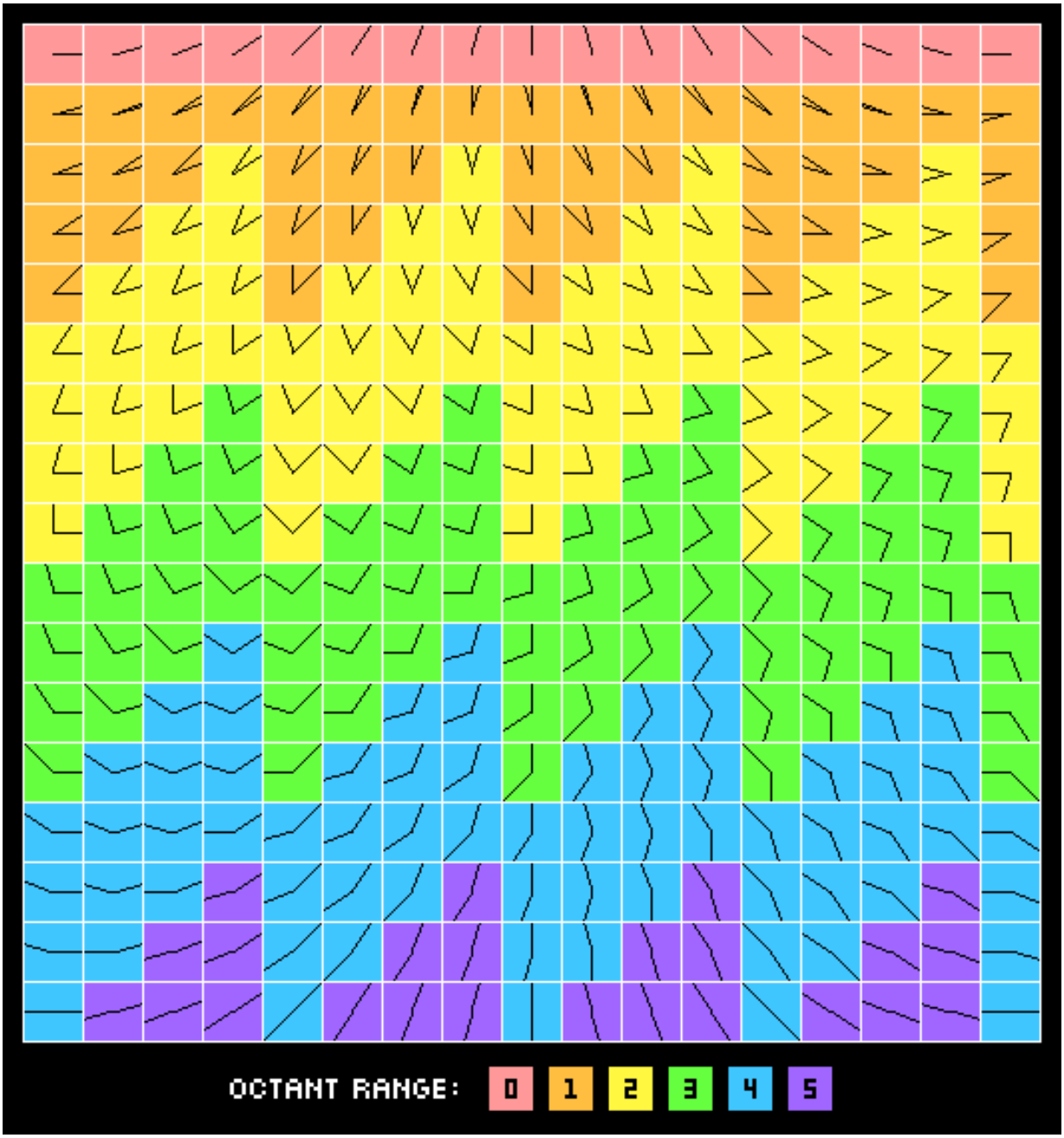


Figure F.4: Angles composed of level 2 edges drawn with the naïve algorithm

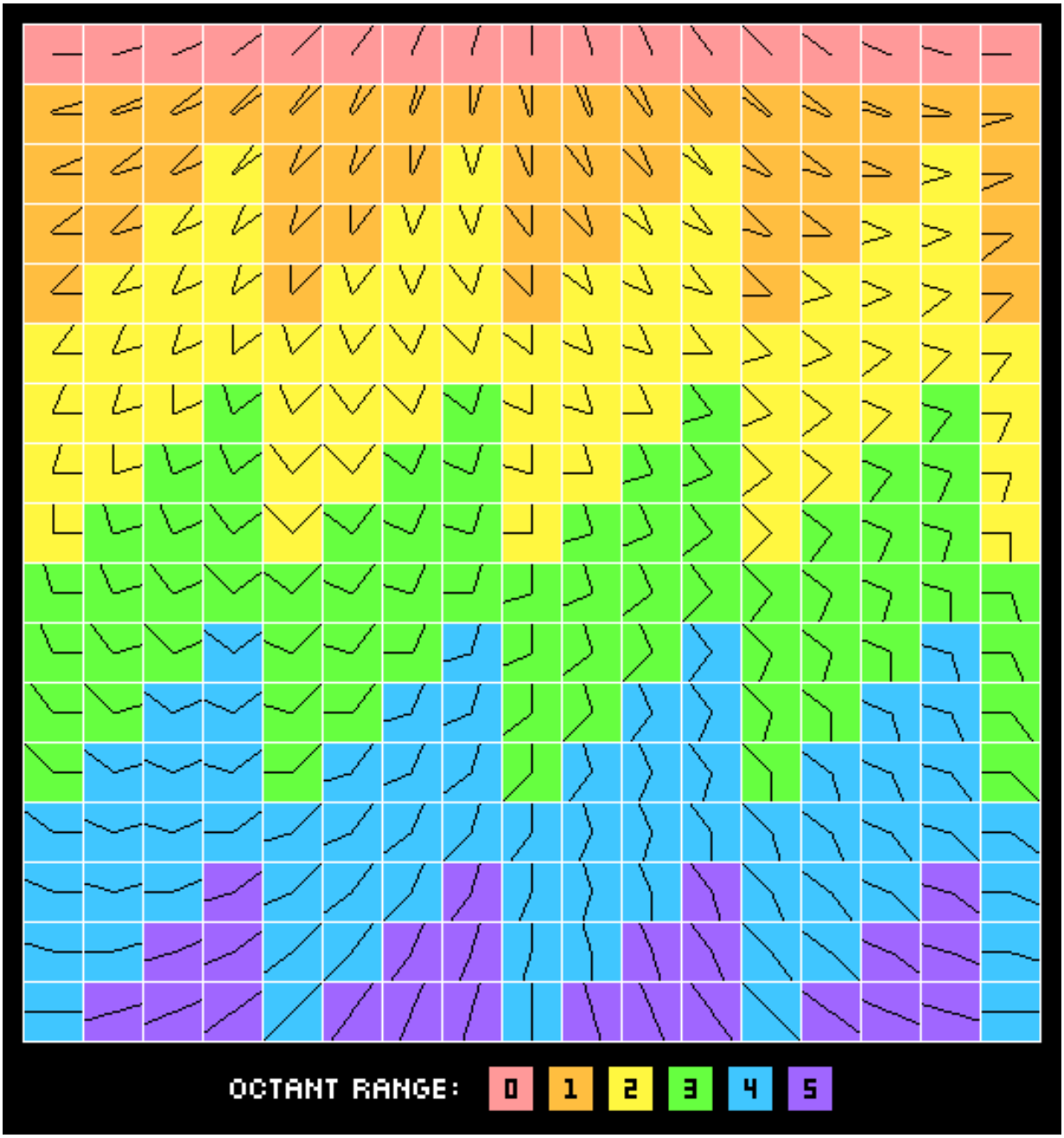


Figure F.5: Angles composed of level 3 edges drawn with our algorithm

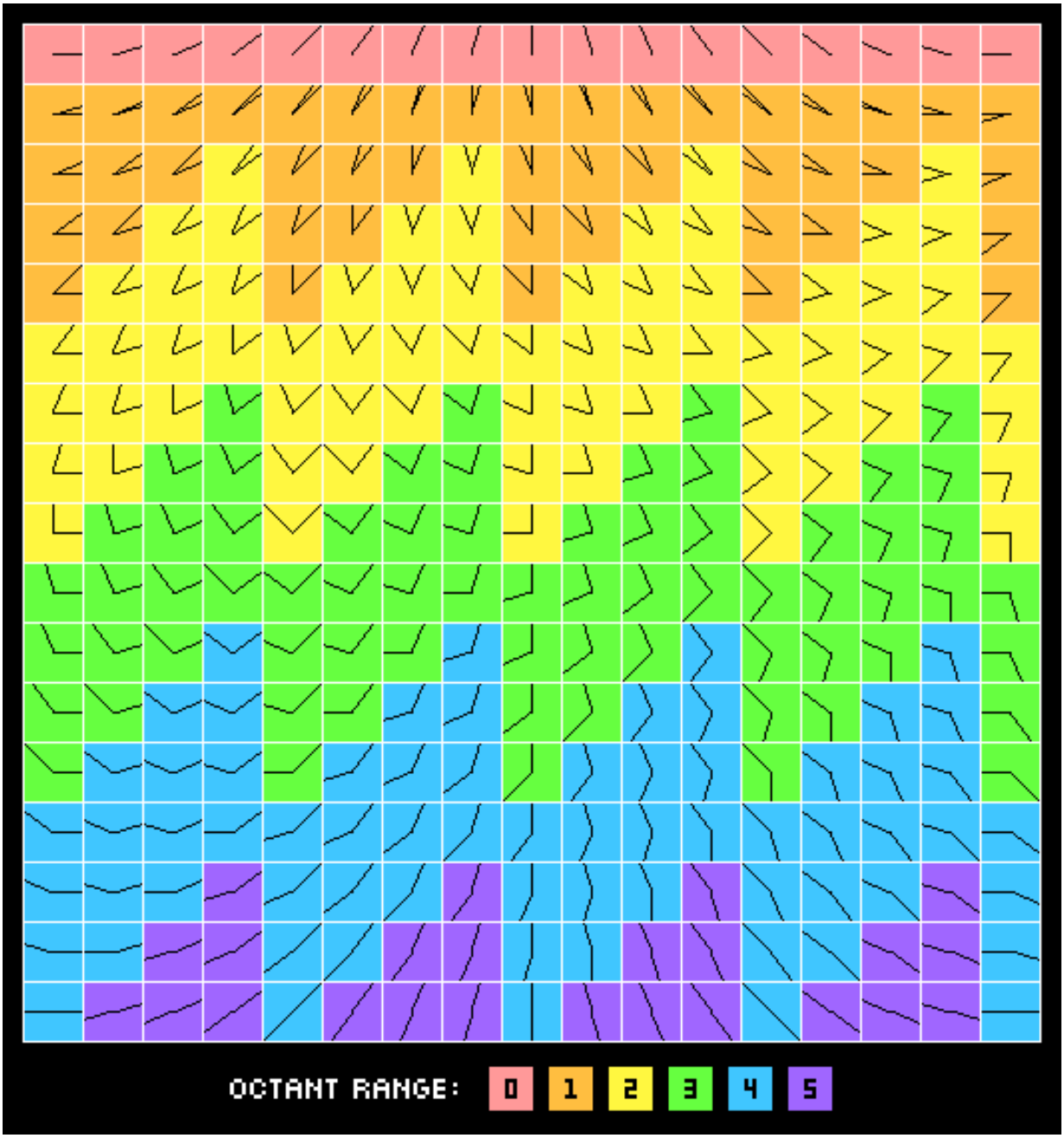


Figure F.6: Angles composed of level 3 edges drawn with the naïve algorithm

References

[1] Bryan D. Ackland and Neil Weste. The Edge Flag Algorithm - A Fill Method for Raster Scan Displays. *IEEE Transactions on Computers*, 30:41–48, 1981.

[2] Adobe Systems Inc. Adobe Illustrator CS5.1. http://www.adobe.com/products/illustrator/, 2013. [Online; accessed 10-Jun-2013].

[3] K. Akeley. Reality Engine Graphics. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 109–116. ACM, 1993.

[4] Narayanan Anantakrishnan and Les A. Piegl. Integer de Casteljau Algorithm for Rasterizing NURBS Curves. *Computer Graphics Forum*, 11:151–162, 1992.

[5] Sten F. Andler. Automatic Generation of Gridfitting Hints for Rasterization of Outline Fonts or Graphics. In *Proceedings of the Interntaional Conference on Electronic Publishing, Document Manipulation, and Typography*, pages 221–23, September 1990.

[6] Apple Inc. *True Type Spec - The True Type Font Format Specification*. July 1990.

[7] Terry L. Benzschewel and Webster E. Howard. Method of and Apparatus for Displaying a Multicolor Image. U.S. Patent #5341153 – Filed: 1988-06-13.

[8] Claude Bétrisey and Roger D. Hersch. Flexible Application of Outline Grid Constraints. In *Raster Imaging and Digital Typography*, pages 242–250. Cambridge University Press, 1989.

[9] V. Boyer and J.J. Bourdin. FastLines: A Span by Span Method. *Computer Graphics Forum*, 18(3):377–384, 1999.

[10] Jack E. Bresenham. Algorithm for Computer Control of a Digital Plotter. *ICM Systems Journal*, 4:25–30, 1965.

[11] Jack E. Bresenham. A Linear Algorithm for Incremental Digital Display of Circular Arcs. *Communications of the ACM*, 20:100–106, 1977.

[12] J.W.S. Cassels. Homogeneous Approximation. In *An Introduction to Diophantine Approximation*, pages 1–3. Cambridge University Press, 1957.

[13] Michael F. Cohen, Jonathan Shade, Stefan Hiller, Oliver Deussen, Wild Tangent, and Technische Universität Dresden. Wang Tiles for Image and Texture Generation. *ACM Transaction on Graphics*, 22:287–294, 2003.

[14] Chris Dannen. The Magical Tech behind Paper for iPad's Color-mixing Perfection. www.fastcolabs.com/3002676, 2013. [Online; accessed 03-May-2013].

[15] Agnè Desolneus, Lionel Moisan, and Jean-Michel Morel. Gestalt Theory and Computer Vision. In *Seeing, Thinking and Knowing*, pages 71–101. Springer Netherlands, 2004.

[16] Alexei A. Efros and William T. Freeman. Image Quilting for Texture Synthesis and Transfer. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 341–346, New York, NY, USA, 2001. ACM.

[17] Candace H. Brown Elliot. Reducing Pixel Count without Reducing Image Quality. *Information Display Magazine*, December 1999.

[18] D. Liauw Kie Fa. 2xSaI: The Advanced 2x Scale and Interpolation Engine. http://www.xs4all.nl/~vdnoort/emulation/2xsai/, 2001. [Online; accessed 10-Jun-2013].

[19] Gerald Farin. *Curves and Surfaces for CAGD: A Practical Guide*. Morgan Kaufmann, 5th edition, 2001.

[20] Francis J. Flanigan and Jerry L. Kazdan. *Calculus Two: Linear and Nonlinear Functions*. Springer, 2nd edition, 1998.

[21] Timothy Gerstner. Pixelated Abstraction. Master's thesis, Rutgers University, New Brunswick, New Jersey, May 2013.

[22] Timothy Gerstner, Doug DeCarlo, Marc Alexa, Adam Finkelstein, Yotam Gingold, and Andrew Nealen. Pixelated Image Abstraction. In *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*, NPAR '12, pages 29–36, Aire-la-Ville, Switzerland, Switzerland, 2012. Eurographics Association.

[23] Timothy Gerstner, Doug DeCarlo, Marc Alexa, Adam Finkelstein, Yotam Gingold, and Andrew Nealen. Pixelated Image Abstraction with Integrated User Constraints. *Computers & Graphics*, 37(5):333–347, 2013.

[24] Rafael C. Gonzalez and Richard Eugene Woods. In *Digital Image Processing*, pages 407–413. Prentice Hall, 3rd edition, 2008.

[25] Jonathan Haber, Sean Lynch, and Sheelagh Carpendale. ColourVis: Exploring Colour Usage in Paintings over Time. In *Proceedings of the International Symposium on Computational Aesthetics in Graphics, Visualization, and Imaging*, CAe '11, pages 105–112, New York, NY, USA, 2011. ACM.

[26] Donald Hearn and M. Pauline Baker. In *Computer Graphics*, pages 68–70. Prentice Hall, 1st edition, 1986.

[27] Donald Hearn and M. Pauline Baker. *Computer Graphics*. Prentice Hall, 1st edition, 1986.

[28] Jeffrey Heer and Maureen Stone. Color Naming Models for Color Selection, Image Editing and Palette Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 1007–1016, New York, NY, USA, 2012. ACM.

[29] Roger D. Hersch. Introduction to Font Rasterization. In J. André and Roger D. Hersch, editors, *Raster Imaging and Digital Typography*, pages 1–13. Cambridge University Press, 1989.

[30] Roger D. Hersch. Font Rasterization, the State of Art. In *Visual and Technical Aspects of Type*, pages 78–109. Cambridge University Press, 1993.

[31] Roger D. Hersch and Claude Bétrisey. Model-based Matching and Hinting of Fonts. In *Proceedings of SIGGRAPH '91*, volume 25, pages 71–80. ACM Computer Graphics, July 1991.

[32] Adobe Systems Inc. *The Type 1 Format Specification*. Addison-Wesley, 1990.

[33] Tiffany C. Inglis. Pixelating Vector Line Art. https://sites.google.com/site/tiffanycinglis/research/pixelating-vector-line-art, 2012. [Online; accessed 14-Mar-2013].

[34] Tiffany C. Inglis. Rasterizing and Antialiasing Vector Line Art in the Pixel Art Style. https://sites.google.com/site/tiffanycinglis/research/rasterizing-and-antialiasing-vector-line-art-in-the-pixel-art-style, 2013. [Online; accessed 14-Mar-2013].

[35] Tiffany C. Inglis and Craig S. Kaplan. Pixelating Vector Line Art. In *Proceedings of the 10th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '12, pages 21–28, 2012.

[36] Tiffany C. Inglis, Daniel Vogel, and Craig S. Kaplan. Rasterizing and Antialiasing Vector Line Art in the Pixel Art Style. In *Proceedings of the 11th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '13, 2013.

[37] Jorge Jimenez, Jose I. Echevarria, Tiago Sousa, and Diego Gutierrez. SMAA: Enhanced Morphological Antialiasing. *Computer Graphics Forum (Proc. EUROGRAPHICS 2012)*, 31(2), 2012.

[38] George H. Joblove and Donald Greenberg. Color Spaces for Computer Graphics. *Computer Graphics*, 12(3):20–25, August 1978.

[39] Donald E. Knuth. *The Metafont Book*. Addision-Wesley, 1986.

[40] Johannes Kopf and Dani Lischinski. Depixelizing Pixel Art. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011)*, 30(4):99:1 – 99:8, 2011.

[41] Johannes Kopf, Ariel Shamir, and Pieter Peers. Content-Adaptive Image Downscaling. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2013)*, 32(6):to appear, 2013.

[42] Yong Kui Liu, Peng Jie Wang, Dan Dan Zhao, Denis Spelic, Domen Mongus, and Bort Zalik. Pixel-Level Algorithm for Drawing Curves. In *Theory and Practice of Computer Graphics*, volume 18, pages 33–40. Eurographics Association, 2011.

[43] Timothy Lottes. FXAA. http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf, February 2009. [Online; accessed 06-May-2013].

[44] J. Manson and S. Schaefer. Wavelet Rasterization. *Computer Graphics Forum*, 30(2):395–404, 2011.

[45] A. Mazzoleni. Scale2x. http://scale2x.sourceforge.net, 2001. [Online; accessed 10-Jun-2013].

[46] Scott McCloud. *Understanding Comics: The Invisible Art*. William Morrow Paperbacks, 1994.

[47] Barbara J. Meier, Anne Morgan Spalter, and David Karelitz. Interactive Color Palette Tools. *IEEE Computer Graphics and Applications*, 24(3), May 2004.

[48] Alexandrina Orzan, Adrien Bousseau, Holger Winnemöller, Pascal Barla, Joëlle Thollot, and David Salesin. Diffusion Curves: A Vector Representation of Smooth-Shaded Images. *ACM Transaction on Graphics*, 27(3):92:1–92;8, 2008.

[49] L. Steinberg R.W. Floyd. An Adaptive Algorithm for Spatial Grey Scale. In *Proceedings of the Society of Information Display*, 17, pages 75–77, 1976.

[50] Peter Selinger. Potrace: A Polygon-based Tracing Algorithm. http://potrace.sourceforge.net, 2003. [Online; accessed 10-Jun-2013].

[51] Alvy Ray Smith. Color Gamut Transform Pairs. *Computer Graphics*, 12(3):12–19, August 1978.

[52] Beat Stamm. Visual True Type: A Graphical Method for Authoring Font Intelligence. In *Proceedings of the 7th International Conference on Electronic Publishing*, EP '98/RIDT '98, pages 77–92, London, UK, UK, 1998. Springer-Verlag.

[53] M. Stepin. Demos & Docs – hq2x/hq3x/hq4x Magnification Filter. http://www.hiend3d.com/demos.html, 2003. [Online; accessed 10-Jun-2013].

[54] International Telecommunication Union. BT. 601: Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide Screen 16:9 Aspect Ratios. http://www.itu.int/rec/R-REC-BT.601, 2013. [Online; accessed 19-Oct-2013].

[55] International Telecommunication Union. BT. 709: Parameter Values for the HDTV Standards for Production and International Programme Exchange. http://www.itu.int/rec/R-REC-BT.601, 2013. [Online; accessed 19-Oct-2013].

[56] Jerry R. Van Aken. Efficient Ellipse-Drawing Algorithm. *IEEE Computer Graphics & Applications*, 4(9):24–35, 1984.

[57] Vector Magic, Inc. Vector Magic. http://vectormagic.com, 2010. [Online; accessed 10-Jun-2013].

[58] Wikipedia. Pixel Art Scaling Algorithms. http://en.wikipedia.org/wiki/Pixel_art_scaling_algorithms, 2013. [Online; accessed 10-Jun-2013].

[59] Freddie Witherden. A Treatise on Font Rasterisation With an Emphasis on Free Software / Combining Anti-aliasing With Hinting. https://freddie.witherden.org/pages/font-rasterisation/#anti-aliasing-with-hinting, 2010. [Online; accessed 10-Jun-2013].

[60] Xiaolin Wu. An Efficient Antialiasing Technique. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, volume 25 of *SIGGRAPH '91*, pages 143–152. ACM, 1991.

[61] Xiaolin Wu. Fast Anti-Aliased Circle Generation. In James Arvo, editor, *Graphics Gems II*, pages 446–450. Morgan Kaufman, 1991.

[62] Alois Zingl. The Beauty of Bresenham's Algorithm. http://members.chello.at/easyfilter/bresenham.html, 2012. [Online; accessed 23-Jun-2013].

[63] Douglas E. Zongker, Geraldine Wade, and David H. Salesin. Example-based Hinting of True Type Fonts. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 411–416. ACM Press/Addison-Wesley Publishing Co., 2000.