A Parameterized Approach for Hedcut Rendering

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Aria Killebrew Bruehl

May 2023

Acknowledgements

This thesis and my ability to make my way through Reed College was not a solo endeavor and could not have been done without the continued support of the following people. Thank you to my mom for telling me in 2018 that I should check out a small liberal arts college called Reed, for every bike ride during Zoom school that stopped me from going insane, and for the endless good advice you have given me, even when it wasn't what I wanted to hear. Thank you dad for always sending me photos of the cat and for being ready with a beautiful home cooked meal the second I'd waltz back through our front door after going too many months without one. Thank you to my twinie, Sophia, for being by my side since the first second of my life and never leaving my side since then. Thank you Chloe for tolerating every moment I was a needy little sister who wouldn't leave you alone, as it turns out that was because I thought you were the coolest (and I still do).

Thank you Jim for suggesting this thesis topic to me! Thank you for all of the support and guidance you gave me over the past eight months.

Katie Maxfield, I simply do not know where to begin. Thank you for the innumerable moments of laughter and joy we have shared over the past two and a half years. Thank you for being there for me in the moments that weren't so joyful, for giving me the encouragement I needed to keep going, and for forcing me to take breaks when that became necessary. And, of course, I thank you endlessly for putting up with me showing you two very slightly different renderings and asking which looked better. Clay, Emma, Jon, and Roan I must thank you for all the coast trips, skiing adventures, lake dips, river swims, brunches, and quesadillas. Clay for the late night PAB dance sessions freshman year. Emma for giving expert advice for how to get through senior year. Jon for every dropped pin on Google maps that brought me to the most beautiful places. Roan for every cookie and baked good that gave me the sugar rush I needed to continue.

Will Howes played an invaluable role in this thesis by sharing with me his install of `Aseprite` and for his constant support and encouragement. Thank you Anna, Dylan, Ivy, Laurel, Mads, Miller, Nathan, and Teagan for every Friday night dinner and for a trip to DisneyLand over winter break that gave me the reset I needed to continue on. Thank you Sky for being my sparring partner and making 4:40 - 5:40 on Tuesdays and Thursdays the perfect opportunity for me to get out all the thesis frustration with punches.

List of Abbreviations

AI	Artificial Intelligence
CIE	International Commission of Illumination
CLI	Command Line Interface
DoG	Difference of Gaussian
LoG	Laplacian of Gaussian
ML	Machine Learning
NPR	Non-Photorealistic Rendering
R&D	Research and Development
RGB	Red, Green, Blue
WSJ	Wall Street Journal

Table of Contents

List of Figures

Abstract

In this thesis I consider the automatic generation of hedcut style portraits using photographs of faces. These renderings mimic those that appear in the Wall Street Journal which are produced by hand with pen and ink. I begin this work by following Kim et al. who render hedcuts by placing stipples so they follow not only the outline of facial features but also isophotes (lines with constant illumination). I apply a variety of image processing techniques to extract facial components and inform stipple placement and size to illustrate tone and depth of the face. I then expand on the work of Kim et al. by introducing an interactive tool that allows for a fully parameterizable version of their approach. Finally I experiment with allowing for areas of negative space and areas of stipple density variation. In this thesis I report on this tool and assess the quality of these methods.

Introduction

Hedcuts, portraits crafted using stipples (tiny dots) and hatches (tiny lines), have become a hallmark of the Wall Street Journal (WSJ) [6]. These portraits are small, typically no wider than a column in a print newspaper, but creating just one portrait can take an illustrator up to five hours to complete [7]. In 2019 the WSJ had a team of five artists who would spend all day crafting these images. Although nothing can replace the look of a hand-drawn image, digital rendering can significantly speed up the process and make this art form more accessible [8].

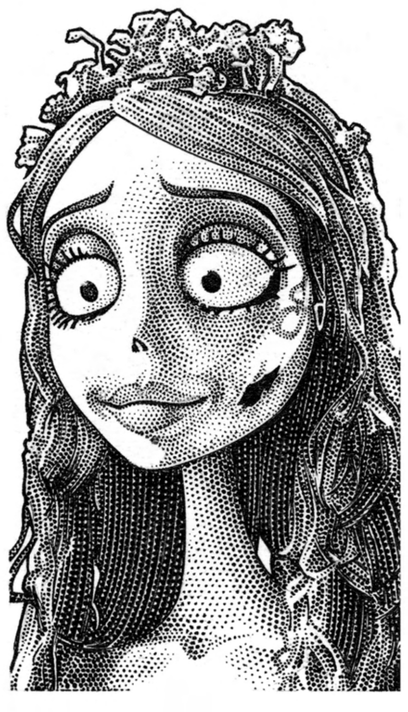


Figure 1: Glass, Randy. Corpse Bride. [Pen and Ink]. Wall Street Journal. Retrieved from https://www.randyglassstudio.com/wall-street-journal-hedcuts.

0.1 What Are Hedcuts

The WSJ once frowned upon the extensive use of photographs in its publications. Editors felt like a heavy use of photographs would distract from the quality and content of the articles [38, p. 356]. Fred Taylor, a past editor, has famously been quoted saying "one word is worth a thousand pictures" [38, p. 356]. So how is it that today, hedcut portraits have become an icon of the WSJ?

Hedcuts first appeared in the WSJ in 1979 when artist Kevin Sprouls presented his stipple and hatching drawings to WSJ. Sprouls' work mirrored that of traditional currency and certificate engravings, for example the portraits shown in Figure 2. Editors felt that this style was more legible than the halftone portraits that were frequently seen in other papers, for example the cat in Figure 3. Halftoned images are made up of black dots placed along a rigid grid with dot size and density reflecting tone. Unlike halftone portraits, the stipples in hedcuts have a flow that follows the contours and surface features of the subject, allowing the portraits to be simple and intentional [38, p. 356]. The WSJ's editors saw a certain sophistication in this style which they deemed compatible with WSJ's aesthetic sensibility [8]

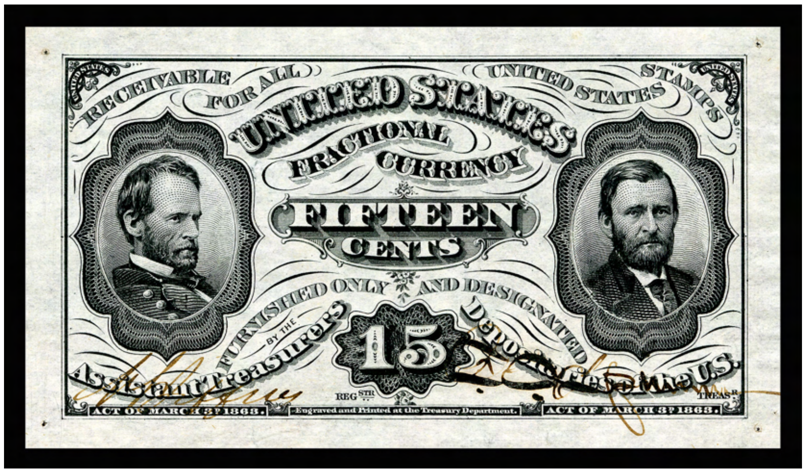

RECEIVABLE FOR ALL UNITED STATES STAMPS
UNITED STATES FRACTIONAL CURRENCY
FIFTEEN CENTS
15
FURNISHED ONLY BY THE ASSISTANT TREASURERS AND DESIGNATED DEPOSITARIES OF THE U.S.
ACT OF MARCH 3D 1863.
Engraved and Printed at the Treasury Department.
ACT OF MARCH 3D 1863.

Figure 2: Civil War era currency engravings from Wikimedia Commons. Image licensed under CC BY-SA 3.0.

Over time the hedcut process has become perfected and the art form is still done by hand with little use of technology. Every day the WSJ's artists print their assigned photograph and trace it in ink with stipples and hatches. The final image is scanned and emailed back to the editors. Although the artists are essentially tracing the photograph the process is still painstaking and deliberate [7]. Just placing dots randomly throughout the face would not do. To capture the flow of a hedcut "you have to know how a face is built, you have to understand the bone structure of the face" [6]. Each dot is placed intentionally, slowly bringing out the contours and shading of the subject's face while remaining simple and minimalistic.

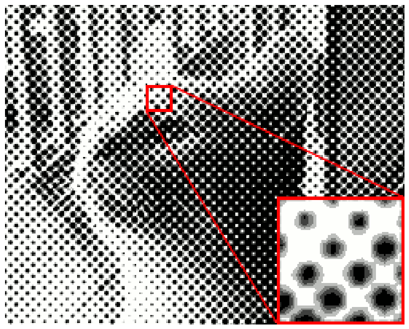


Figure 3: A halftoned cat from Wikimedia Commons. Image licensed under CC BY-SA 3.0.

0.1.1 The Core Qualities of a Hedcut

Although there is slight variation in style amongst different hecut artists I have identified a number of "core qualities" that are found in nearly every hedcut I examined. This list is by no means final, however I found it useful for evaluating results later on. These qualities are as follows:

Hedcuts...

- ☐ are small [7]
- ☐ are portraits of people or animals[1]
- ☐ only include the subject's face without the background
- ☐ are two tone (black and white) images
- ☐ use the size of stipple dots and the thickness of hatch lines to depict tone
- ☐ use stipple dots to shade the subject's face [8]
- ☐ use hatch lines to shade the subject's clothing
- ☐ place stipples so they align with facial features [38]
- ☐ do not place dots in the highlights of a subject's face
- ☐ use denser stippling for key features such as a subject's eyes, nose, and lips
- ☐ use strokes for hair
- ☐ outline the subject with strokes

[1]WSJ uses a stippled style to illustrate their daily feature known as the "A-hed" which can include broader scenes rather than just portraits [8]. For this thesis, however, we will only consider hedcuts to be of people and animals.

0.2 Non-Photorealistic Rendering

In the earlier days of computer graphics, the driving force behind the field was photorealism [17, p. 1]. However impressive photorealism is, it is not always the best way to convey visual information to the viewer. Gooch and Gooch [17, p. 1] use the example of a sailboat to describe this phenomenon. A photorealistic image of a sailboat allows the view to infer a vast amount of information about the scene: the time of day, weather, speed of the boat, etc. They explain, however, that "such an image would be little use to someone attempting to build a sailboat" who would "certainly prefer technical drawings or blueprints, while someone who simply wanted to communicate the idea of a sailboat may only need to draw a shape representing the boat and a triangle representing the sail" [17, p. 1]. Renderings that do not aim to create photorealistic animations or images and that intentionally omit a certain level of detail are called non-photorealistic renderings (NPR). Digitally rendered hedcuts are a form of NPR.

0.2.1 General Motivation for NPR and This Thesis

NPR images and animations are able to concisely and effectively communicate an idea by harnessing the techniques traditionally used by artists to emphasize features or a scene, expose minute attributes, or omit extraneous information. In the case of hedcuts, this means omitting features like color or the fine grain aspects of facial texture and emphasizing the lines and curves of one's face. Understanding the motivations behind NPR is critical for designing new techniques and informing the practical application of the field. In a survey of image and video-based artistic stylisation [20] Isenberg overviews a variety of studies on the effectiveness of NPR.

Schumann et al. [35] examined the use of NPR in architectural renderings and found that the style of a rendering can have an effect on how willing someone is to interact with the depiction [20, p. 314]. Duke et al. [13] examined the use of NPR in psychological principles and Harper et al. [18] focused on aspects of environmental psychology and how level of detail in a rendering affects people's behavior. Both researchers found that the control in the amount of detail in NPR can be used to effectively guide people's attention and emotions [20, p. 316]. These studies and others like them motivate the continued development of NPR for effective communication of designs, directing peoples attention to certain aspects of a figure, and the effect these renderings can have on emotion [20, p. 317].

My work has similar motivations. Hedcuts are an efficient method for portraying the subject while directing the viewers' attention to the flow and alignment of facial features. Additionally, I wanted to take on an NPR thesis because doing so would give me the opportunity to experiment in the field of image processing while building on my existing knowledge of computer graphics. Finally, I am not an artist, however, this thesis gave me the opportunity to explore computer generated art. The difficulties I encountered in rendering hedcuts gave me an appreciation for the work of hedcut artists and a deeper understanding of the knowledge of the human face that is necessary for this style.

0.3 Generating Hedcuts with and without Artificial Intelligence and Machine Learning

In recent years various labs such as the WSJ's Research and Development (R&D) team have built Artificial Intelligence (AI) models that quickly[2] render hedcut style portraits based on an input image. The WSJ's R&D team trained their model by "[running] ever more data points [portraits of people and hand-drawn hedcuts] through [the] model, [providing] feedback on the machine's performance and [watching] it get progressively better" [8]. The training process involved hand-tagging over 2,000 photos so the machine could learn to recognize precise aspects of a person's face.

The WSJ team had to overcome many challenges such as teaching the tool to distinguish the subject from its background, to use hatches for clothing and stipples for skin, and to properly render bald people. They also overcame the challenge of "overfitting" which happens when an AI fits a limited set of data too closely. In the case of the R&D team's hedcut AI, the result of overfitting was disturbing portraits like those seen in Figure 4. Although WSJ's AI has done an excellent job rendering hedcuts, see Figure 5, it is also possible to render these images without AI or ML.

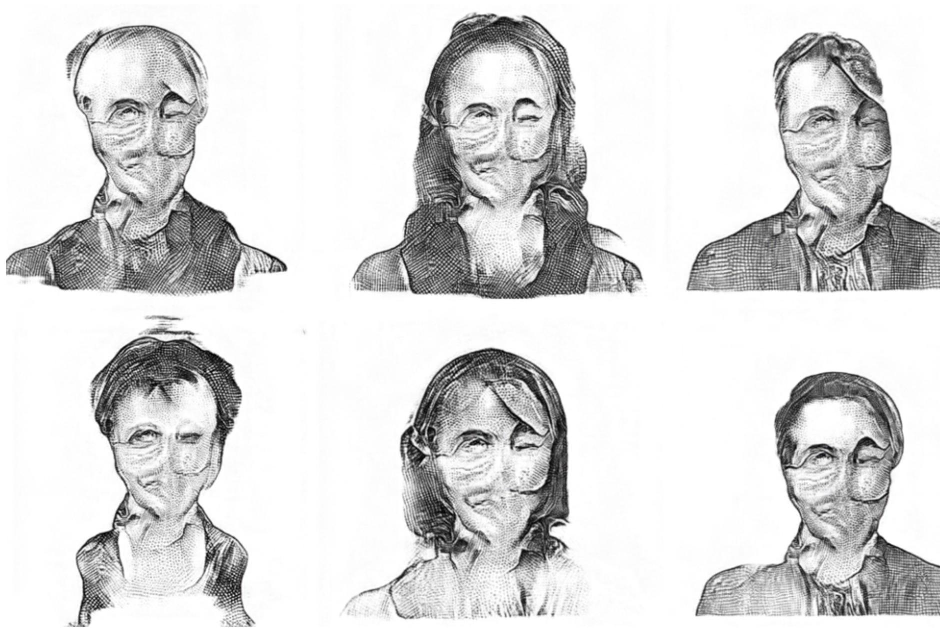


Figure 4: Early AI rendered hecuts from Wall Street Journal. Used with permission from The Wall Street Journal, WSJ.com. Copyright 2019 Dow Jones & Company, Inc. All rights reserved.

When generating hedcuts without AI or ML elements of computer vision and image processing are combined to extract the necessary facial features from images and mathematical calculations are used to determine the shape, size, and placements of stipple dots [36, 25]. In this thesis I will be surveying these techniques, combining

[2]In 2019 the AI was able to render portraits in about 90 seconds [8].

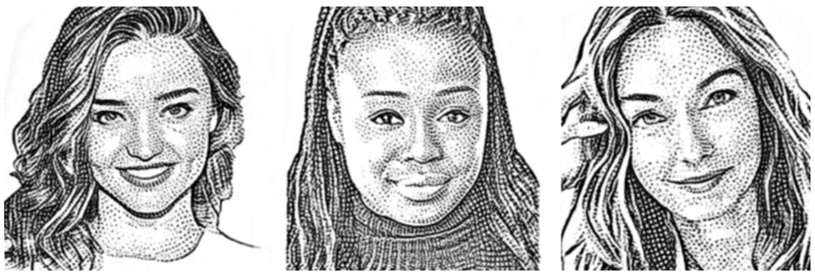


Figure 5: AI rendered hecuts from Wall Street Journal. Used with permission from The Wall Street Journal, WSJ.com. Copyright 2019 Dow Jones & Company, Inc. All rights reserved.

them in novel ways, and experimenting with new rendering techniques with the goal of generating "better" looking results.

0.3.1 Related Work

There have been a diverse range NPR approaches for illustrating faces. The goal of these works is to produce images that are simple but still identifiable as the person in the original photograph [33]. The style of NPR renderings varies greatly. Kasao and Miyata [22] propose an algorithm that can produce various styles of paintings from source photos, including paintings in the style of specific artists such as Van Gogh. Rosin and Lai [33] also propose a NPR technique in the style of a specific artist, in this case Julian Opie, part of the New British Sculpture movement. Rhee and Lee [31] propose a method for cartoon-like avatar generation that closely mimics the style of Wii avatars.

Using NPR to mimic broader styles is also common. Ostromoukhov [29] introduces basic techniques for digital facial engraving to imitate traditional copperplate engraving. This style is similar to hedcuts in that the subject is rendered using black and white lines that follow a general feature flow. There have been many approaches that apply NPR techniques for rendering hedcuts. Kim et al. [24] propose a stippling method where dot placement is guided by a feature flow extracted from feature lines. Son et. al [36] use a structured grid for directional stippling that aligns with feature lines in both the parallel and perpendicular directions. Kim et al. [25] consider isophotes (lines with constant illumination) to better capture the perceptual cues of the input image.

0.4 The Ethical Question

Before we begin I must address the ethical question of machine rendered art, as it has been a topic of much debate in recent news. The full scope of this discussion is outside the bounds of this thesis introduction. Nonetheless I will attempt to present some of the most basic questions surrounding this topic.

These ethical debates typically center around AI generated art and the question of whether or not an AI's renderings fall under copyright violation. An AI that produces art will learn many different artistic techniques and styles. These AI are built by "scraping millions of images from the open web, then teaching algorithms to recognize patterns and relationships in those images and generate new ones in the same style" [32]. The images an AI is trained on are never included in the final product but users can direct an AI to render images in a particular style [9]. This could include producing images that look like watercolor paintings or charcoal drawings. AI can also be trained to produce images that replicate a specific artist's unique style. The developers of these AIs rarely ask the artists if their works can be used to train their AI. As a result an artist who uploads their art to the internet with the hope of promoting their work may in fact be aiding an AI that could become their competitor. This creates a slippery slope between AI being "inspired" by the great artists of our present and our past and them "stealing" artists' unique styles. Since the field of AI art is so new lawmakers do not have a clear answer to whether or not AI training is violating copyright law.

I believe that my thesis and the work of others who attempt to render a particular artistic style without the use of AI are being "inspired" by these artists rather than "stealing" their work. Just like an AI attempting to mimic the hedcut style I will be looking at images found on the open web for inspiration. However,I find that the scale on which this is done creates a notable distinction between work done with AI and work done without it. An image processing AI will be trained on massive amounts of data, thousands of images, nonstop. In my work and others like it that type of data processing is impossible. In NPR without AI and ML, the programmers carefully examine example images, make an attempt at mimicking them, compare the output to the original images, and repeatedly tweak numerical parameters and redesign algorithms until they reach an output that looks "right". This process is slow and tedious and is limited by the amount of time the programmer is willing to spend on their work[3].

0.5 Process Overview

In this thesis I will present a tool I built that attempts to mimic aspects of hedcut rendering. This tool takes photographs as input and uses image processing and computer vision techniques to output an image that mimics the hand-drawn hedcut style. To complete this work we closely followed the process defined in Kim et al. [25]. This process works as follows:

1. Extract the "feature lines" from the image by detecting edges and isophotes.
2. Use the feature lines to generate a distance map.
3. Use the distance map to generate an offset map.

[3]I, for example, was limited by my need to get eight hours of sleep a night and by the number of chocolate covered espresso beans I felt was healthy to consume over a 9 month period.

4. Use the offset map to initially place dots.

5. Use the offset map and the feature lines to adjust the initial dots.

6. Use the input image and adjusted dots to render the final image.

Figure 6 shows an example input image to our program. Figure 8 shows the intermediary steps in the process. Note that throughout this thesis any figure marked with †has been modified for legibility purposes. In most cases this means rendering larger dots than the actual image so they are visible to the reader. Figure 8a shows the final hedcut rendered with this process. Note that in Chapters 2 and 3 we will use Figure 6 as our input for consistency. Appendix C shows results with various input images. For this process I expanded on the work of Kim et al. [25] with the algorithm used to find the distance between pixels in an image and in the edge detection process. After getting initial results I experimented with implementing a completely parameterized approach, rendering with negative space, and rendering with dot density variation, as seen in Figure 8b and Figure 8c, respectively.

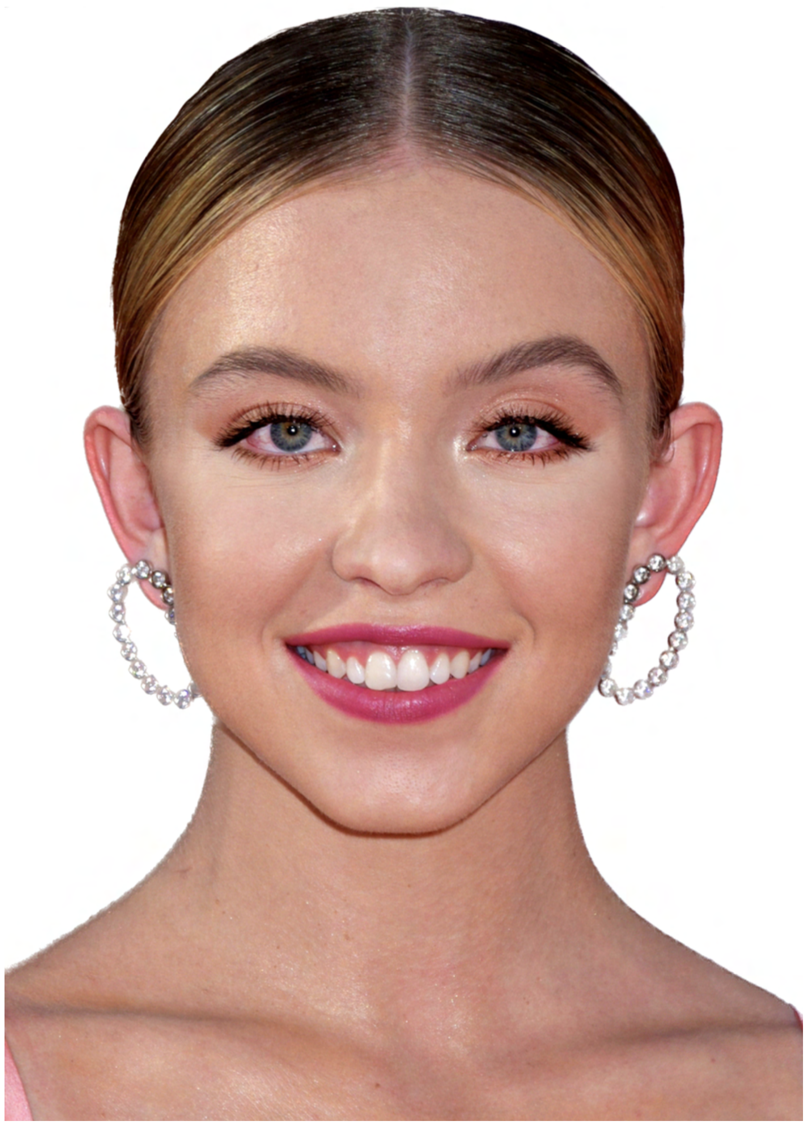
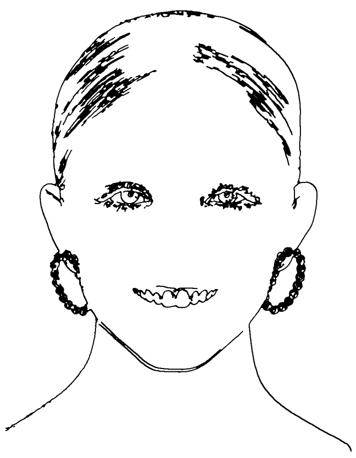
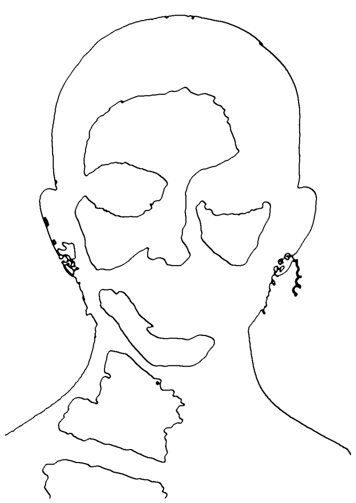
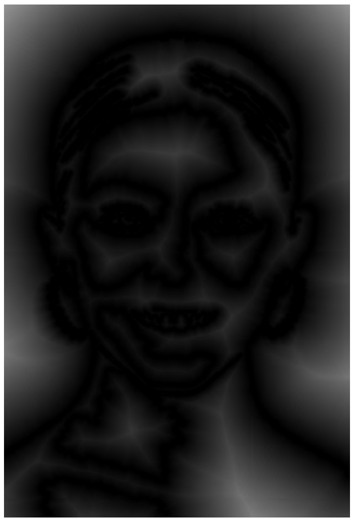
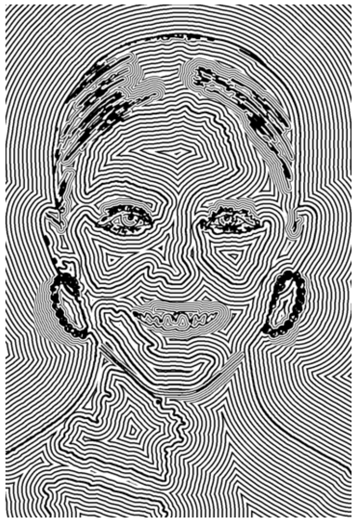


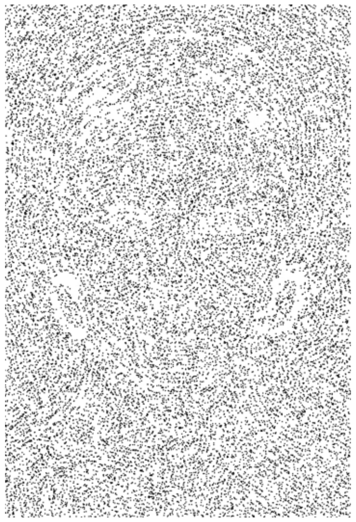
Figure 6: Actress Sydney Sweeny at the **Once Upon a Time in Hollywood** premier from Wikimedia Commons. Image licensed under CC BY-SA 4.0.


(a) Step 1, edges.


(b) Step 1, isophotes.


(c) Step 2, distance map.


(d) Step 3, offset map.


(e) Step 4, initial dots†.

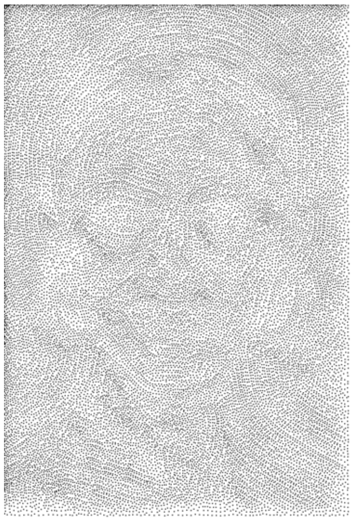
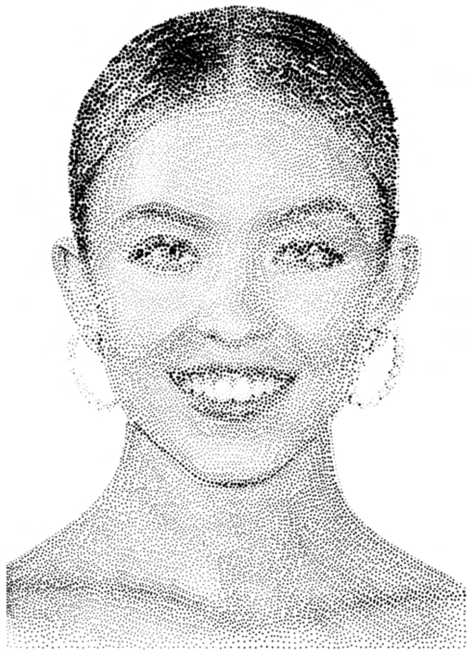
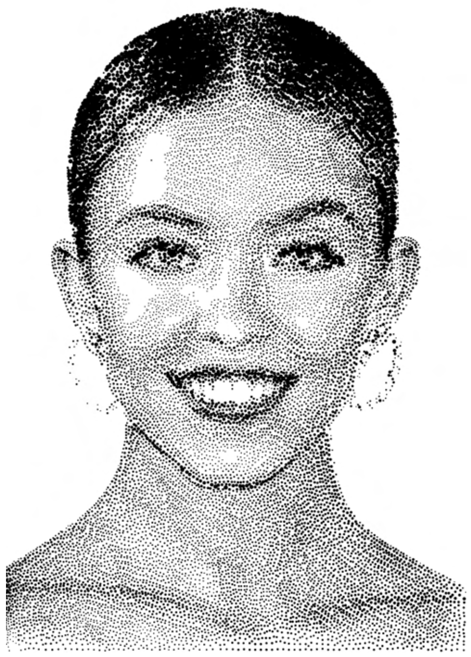

(f) Step 5, adjusted dots†.

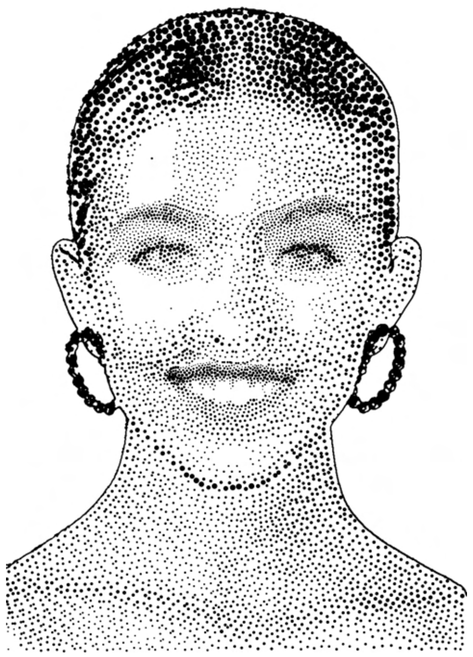
Figure 7: The intermediate steps for hedcut generation.



(a) Initial rendering of Figure 6.



(b) Rendering of Figure 6 with negative space.



(c) Rendering of Figure 6 with density variation.

Figure 8: Three final renderings of Figure 6.

Chapter 1

Background

1.1 Representing Images on a Computer

The images that we see everyday in newspapers, on billboards, and in works of art are made up of different colors and shapes applied to a physical media. A painter puts paint on a canvas, a laser jet printer puts ink on pages, a toddler uses crayons to add flair to their bedroom wall, etc. If we wanted to tell an artist how to reproduce Figure 1.1a we might tell them to take a black piece of paper, paint a white line down the middle of it, and another across the center of it. If we wanted to store a digital version of this image, however, we could not tell our computer to "take out a piece of paper" and then "paint" lines on that paper. This is because our computer does not have any paper, nor does it have any paint! Instead our computer has memory and in that memory it can store numbers.

A digital image is stored in a computer memory as a *matrix* of numerical values. Figure 1.1a, for example, can be divided into 25 *pixels*, 5 in each row and 5 in each column. A computer would represent this image as a 5×5 matrix storing the values of the 25 pixels in the image, as shown in Figure 1.1b. In this image a pixel either has the value 0 for black or 1 for white. Since a pixel can only take on 2 values this is a *binary image*.

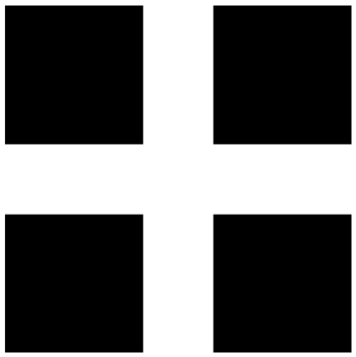
Definition 1. In computer vision and image processing a *matrix* is a two-dimensional array of numerical values with the same dimensions as the image it represents.

Definition 2. A *pixel* is a small area in an image to which a numerical value is assigned [28].

Definition 3. A *binary image* is an image that only has two gray levels [11].

1.2 Channels, Depth, and Color Spaces

To represent and manipulate an image in a more complex way we need to store more information at each pixel than just a 0 or 1. We will now introduce the notion of pixel "channels" and channel "depth". Then we will show how these factors combine to give us a sense of the color space of an image.



(a) A very simple image.

```
[ [0, 0, 1, 0, 0]
  [0, 0, 1, 0, 0]
  [1, 1, 1, 1, 1]
  [0, 0, 1, 0, 0]
  [0, 0, 1, 0, 0] ]
```

(b) The corresponding representation as numbers

Figure 1.1: The pictorial and numerical representations of an image.

1.2.1 Channels

The number of channels in an image corresponds to the number of values needed to represent each pixel. Recall that in binary images each pixel has a value of either 0 or 1. In a grayscale image each pixel corresponds to a real number value representing a different shade of gray. Binary and grayscale images can be represented with only a single channel since each pixel only needs to store one value.

In a color image each pixel must hold more than one piece of information. In a red, green, blue (RGB) color image, for example, each pixel holds a triple of values representing its red, green, and blue components. This triple of values corresponds to the three channels of each pixel, often called color channels. The value in each channel represents the amount of red, green, and blue in the pixel. Section 1.2.3 explains the RGB color space in more detail. Figure 1.2 shows the three color channels of an RGB image. Figure 1.3 shows an image rendered in three ways. Figure 1.3c is a binary representation. Figure 1.3b is a grayscale representation where pixels have one channel that holds a range from 0 to 255, Section 1.2.2 explains the significance of the value 255. Figure 1.3a is a RGB color representation where pixels have three channels that each hold a range from 0 to 255.

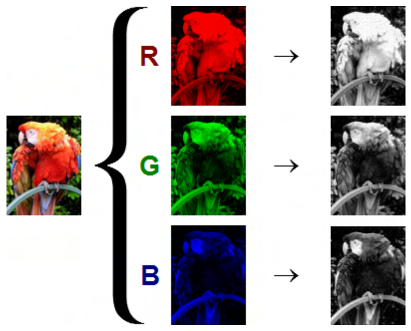
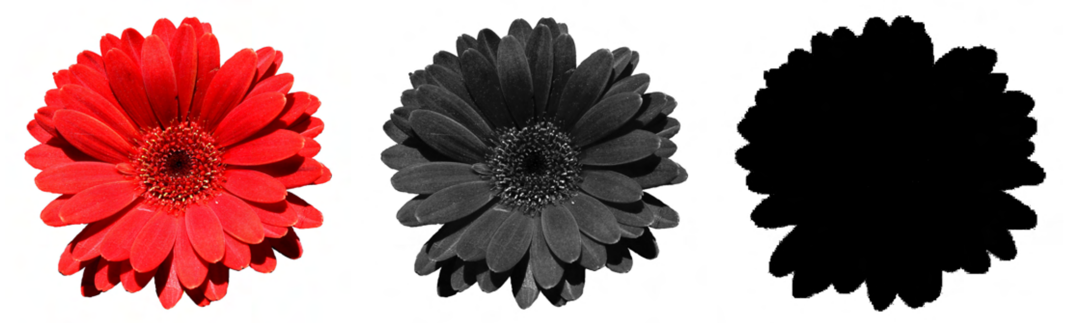


Figure 1.2: RGB channels from Wikimedia Commons. Image reprinted as part of the public domain.



(a) A colored flower. (b) A grayscale flower. (c) A binary flower.

Figure 1.3: A colored, grayscale, and binary flower from Wikimedia Commons. Image reprinted and altered under CC BY-SA 2.5.

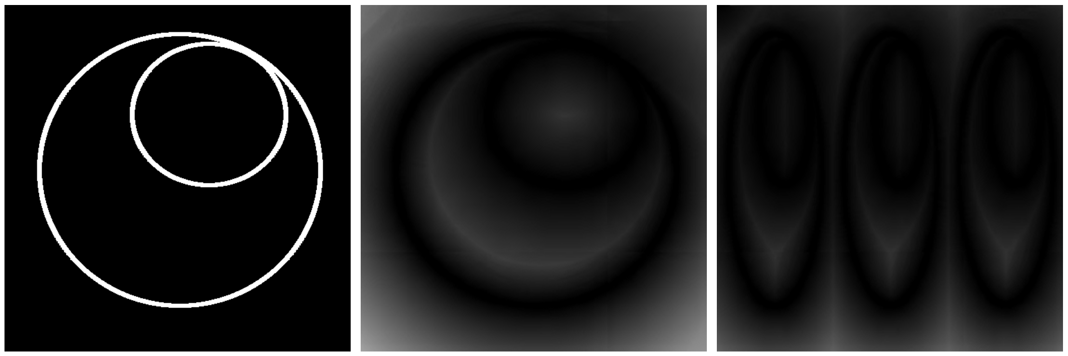
1.2.2 Depth

The depth of a channel corresponds to the type of data stored in the channel. In Figure 1.4 all the matrices have depth `8U` meaning each channel stores an unsigned 8-bit integer giving a range from 0 to 255. Using 8-bits per channel is typical since it does not take up too much memory but still allows for good color range. There are times, however, when more space is needed. For example when generating distance maps as described in Section 3.1 I needed images with channel depth of `32S` meaning each channel holds a signed 32-bit integer in the range -2147483648 to 2147483647. It is also possible to have a depth that is a floating point value rather than an integer value. Using a floating point depth is common when values get normalized to be in the range 0 to 1 and fractional values are necessary.

Throughout my thesis, awareness of the number of channels in an image and the depth of each channel was crucial. At the start of my coding process I encountered bugs due to manipulating an image as if it was a three channel image when it was in fact a single channel image. Figure 1.4b and Figure 1.4c are the results of running the distance map algorithm, explained in Section 3.1.1, with Figure 1.4a as input. Figure 1.4a is of type `CV_8UC3` meaning there are 3 channels with depth of `8U`. In Figure 1.4b the matrix that the output was being written to was also of type `CV_8UC3`. In Figure 1.4c, however, the output was written to a matrix of type `CV_8UC1` meaning it has only one channel resulting in the image being squashed. In other cases the wrong type of output matrix would result in seg-faults and the program crashing before it could generate output.

1.2.3 RGB color space

The idea of the RGB color space closely follows the principles of human vision. The retinas in our eyes contain light receptor cells called rods and cones. Rods detect light in low-light situations (at night) and cones detect light in greater light situations (during the day). There are three types of cones which sense different wavelengths of light, long, medium, and short. They are often referred to as “red”, “green”,



(a) Input image of type CV_8UC3. (b) Output written to CV_8UC3 image. (c) Output written to CV_8UC1 image.

Figure 1.4: A bug due to an image matrix having the wrong type.

and "blue" receptors due to the range of wavelengths they detect [19, p. 749]. The combination of the responses from the cones generates a sensation of color making human vision an *additive* color model. Yellow, for example, is an "invented" color. The brain processes and combines the information detected by the red and green cones to perceive the color yellow.

Definition 4. *Additive* color is the process of adding different wavelengths of light to create a specific color. The visible light spectrum's primaries (red, green, and blue) are mixed in various quantities to produce secondary colors [30].

In the RGB color space, each pixel holds three values. The first corresponds to the red component of the color, the next to the green, and the last to the blue[1]. Like human vision the RGB color model is an additive color model meaning we start with black (0) and add red, green, and blue components to produce the spectrum of color. We can thus represent the RGB color space as a cube where the X, Y, and Z axes correspond to the range of red, green, and blue, as seen in Figure 1.5.

In Figure 1.5 we can see that black is represented as the triple $(0, 0, 0)$ meaning no color is added to any channel and white is represented as the triple $(255, 255, 255)$ meaning the maximum amount of color added to each channel. Pure red is represented as the triple $(255, 0, 0)$, pure green as $(0, 255, 0)$, and pure blue as $(0, 0, 255)$[2]. We can combine different amounts of these three primary colors to generate other colors such as magenta which is equal parts pure red and pure blue, $(255, 0, 255)$. If we wanted

[1]The image processing library `OpenCV` uses the BGR color space for color images by default meaning pixels store the blue component first, then the green, then the red. This was quite surprising to me at first since the vast majority computer graphics and image processing libraries and tools use RGB. In the early days of `OpenCV` the BGR format was popular among camera manufacturers and software providers [27]. This popularity meant `OpenCV` used BGR as their default color space. This anomaly does not result in any bugs since `OpenCV` will read in images unchanged, hence maintaining the RGB format.

[2]Note that in this representation of RGB color space the channel depth is a 8-bit unsigned integer. The channel depth can be changed but we can still make pure red, green, or blue by giving the respective channel its maximum value and setting the other channels to 0.

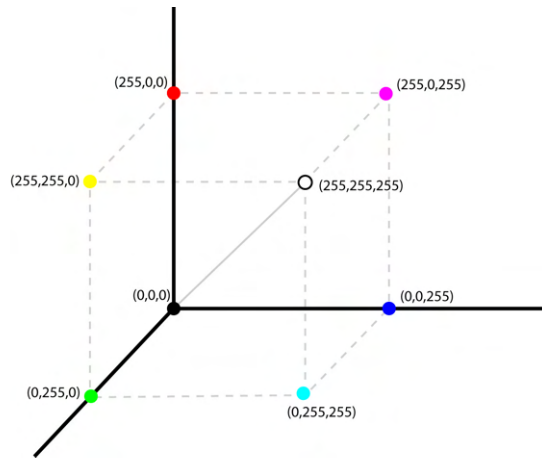


Figure 1.5: RGB cube from Wikimedia Commons. Image reprinted under CC BY-SA 3.0.

something closer to violet we would decrease the amount of red and blue and add a little bit of green, giving a value like $(64, 25, 127)$. We can still represent all the shades of gray in the RGB color space. The possible shades of gray lie on the diagonal between black, $(0, 0, 0)$ and white, $(255, 255, 255)$. Thus a gray value will have equal values for the red, green, and blue components.

1.2.4 CIE color space

RGB and BGR are by no means the only way to define color. Printers, for example, use the CMYK, Cyan-Magenta-Yellow and black (K), model of color[3] and U.S. commercial television broadcast uses the YIQ model which was designed so televisions could broadcast a signal for both black-and-white and color televisions while using bandwidth efficiently [19, p. 774, 775]. In 1931 the International Commission of Illumination (CIE) defined three standard primaries, X, Y, and Z which together form a triangle that encompasses all possible sensor responses [19, p. 763]. These primaries have the following advantages:

1. The Y primary is exactly the luminous efficiency for a spectral light source. Thus if we have a light source T written as $T = c_x X + c_y Y + c_z Z$ the coefficient c_y corresponds to the perceived intensity of the light [4].

[3]The CMY color model is a *subtractive* color model meaning colors are created by removing wavelengths. This process is similar to what an artist would do when mixing paints. To create green, for example, an artist would mix blue and yellow. When the paints are mixed the resulting color is the wavelengths that both paints reflect [30]. In the CMYK model K, pure black, is needed because mixing equal parts cyan, magenta, and yellow doesn't quite create pure black [19, p. 775].

[4]This was significant in developing black-and-white televisions. The signal needed to provide some sort of Y component of the lights that the camera was perceiving. When color signals began being broadcast as well, the c_x and c_z components, which define color, were sent in a different band which black-and-white televisions would ignore [19, p. 763].

2. The color matching functions for X, Y, and Z are everywhere nonnegative so all colors are expressed as non-negative linear combinations for the primaries.

3. Then red, green, and blue primaries can be identified as points in the XYZ-space so any color that is found in the RGB-space can be found in the XYZ-space with a direct conversion.

The first and third points were of particular importance for this thesis. At times it was necessary to convert between RGB and CIE spaces which is possible due to point three. Other times it was necessary to examine the luminosity of a pixel which could be done using a pixel's Y component. In particular, this thesis uses the $CIEL*a*b*$ color space as seen in Figure 1.6. The $L*$ component captures the notion of intensity and the $a*$ and $b*$ components encode chromaticity, $a*$ gives relative red-green and $b*$ gives relative blue-yellow [19, p. 769]. In Figure 1.6 we can see that as the L component decreases the colors become less intense.

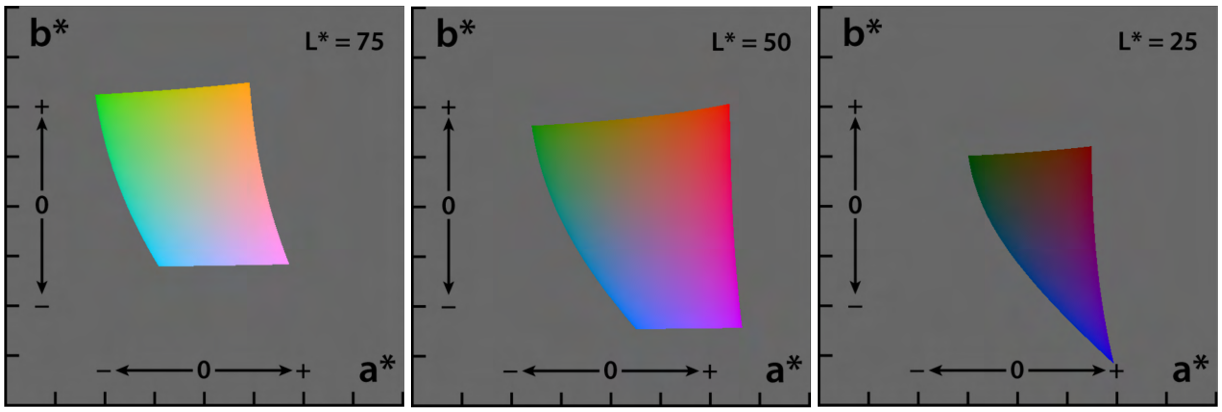


Figure 1.6: The CIE 1976 L*a*b* color space showing only colors that fit within the standard RGB space (displayed on a typical computer monitor) from Wikimedia Commons. Image reprinted and altered under CC BY-SA 3.0.

1.3 Tech Stack

The primary technologies used for this thesis were `OpenCV` with C++ and `Aseprite`. `OpenCV` provided me with a robust framework of image processing functions ranging from the basic operations (reading in and saving images, converting an image from color to grayscale and vice versa, applying thresholds etc.) to more complicated filters and algorithms (Difference of Gaussians, Canny edge detection, bilateral filters, connected component detection, etc.). I give a brief overview of the more complicated functions I used in Section 1.4. I opted to use `OpenCV` with C++ rather than Python since C++ programs often execute faster. Image processing often requires looping over every pixel in the image at least once. If an input image has n rows and n

columns an algorithm that looks at each pixel (without the use of parallelism[5]) would take $O(n^2)$ time[6]. Using language systems other thanC++ would likely have less acceptable performance.

`Aseprite` was used to create every simple input image you see in this thesis, for example Figure 1.4a. Being able to create minimal input images was essential for debugging my work. When testing and debugging code a programmer will run their work with the simplest possible input. In my case these were binary images that were only a few pixels wide.

1.4 Overview of Library Functions Used

The majority of the library functions I used were some type of filter. Filters modify or enhance an image through neighborhood operations. That is, the output value of a given pixel is calculated with respect to the values of the pixels in its *neighborhood*. Often filters will use a kernel to sample the values of the pixels in the neighborhood. A kernel is simply a matrix of values that is placed on each pixel in the input image. The value for p in the output image is calculated using the kernel and the pixels in p's neighborhood.

Definition 5. A pixel's *neighborhood* is a set of pixels, denoted N, defined by their proximity to a given pixel, p [4].

Say we wanted to create a simple "shift-left" filter which takes an image as input and outputs the same image but shifted to the left by one pixel. We will define the neighborhood, N, of a pixel, p, as p and the eight pixels touching p. The kernel used for this operation is shown in Figure 1.7.

0	0	0
0	0	1
0	0	0

Figure 1.7: A simple kernel

Suppose we use Figure 1.8a as our input image. To generate our output image we will place the center of our kernel, the shaded cell in Figure 1.7, on each pixel, p, of the input image. When we do so each pixel in p's neighborhood, $q \in N$, will have a cell of the kernel on top of it. We will calculate

$$s = \sum_{q \in N} q.value \cdot k[q]$$

Where $q.value$ is the value of pixel q and $k[q]$ is the kernel value at pixel q when the kernel is centered at p. Then we will set the value of p in the output image to be s.

[5]Parallelism is a technique that makes programs faster by performing several computations at the same time.

[6]An $O(n^2)$ algorithm (quadratic time algorithm) will execute in time proportional to the square of the size of the input.

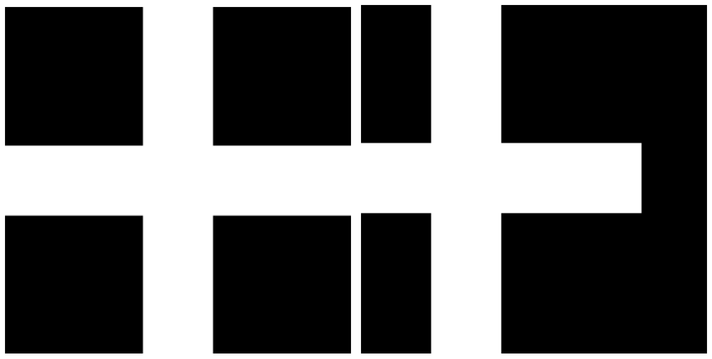

(a) The input to the "shift-left" filter. (b) The output of the "shift-left" filter.

Figure 1.8: The result of applying a simple filter to a simple input image

To apply the "shift-left" filter to Figure 1.8a we first place the center of the kernel on pixel $(0, 0)$, as shown in Figure 1.9a. We calculate the value of pixel $(0, 0)$ in the output image as:

$$s = (0 \times 0) + (1 \times 0) + (0 \times 0) + (0 \times 0) = 0$$

Then we will give pixel $(0, 0)$ a value of 0 in our output image. Now we move our kernel to pixel $(0, 1)$, as seen in Figure 1.9b, and get the output value as

$$s = (0 \times 0) + (0 \times 0) + (1 \times 1) + (0 \times 0) + (0 \times 0) + (0 \times 1) = 1$$

Then we will give pixel $(0, 1)$ a value of 1 in our output image. Note that in each of these cases only part of the whole kernel was placed on the image, which is allowed. If we were to calculate the output value for pixel $(4, 1)$ we would move our kernel to pixel $(4, 1)$, as seen in Figure 1.9c, and get the output value as

$$s = (0\times 1)+(0\times 1)+(0\times 1)+(0\times 0)+(0\times 0)+(1\times 1)+(0\times 0)+(0\times 0)+(0\times 1) = 1$$

Then we will give pixel $(4, 1)$ a value of 1 in our output image.

After applying the "shift-left" filter to every pixel we will have our complete output image which is the simple white cross shifted to the left by 1 pixel as seen in Figure 1.8b.

1.4.1 Bilateral Filtering

A bilateral filter smooths an image thus removing noise while preserving edges. Other approaches to filtering assume a slow spatial variation in pixel values. It is assumed that pixels near each other have similar values and can therefore be averaged together [37]. This results in blurred edges which is not acceptable for this work. Bilateral filters, in contrast, only average together perceptually similar colors. The bilateral filter can be told explicitly which colors are similar and which are not [37].

Bilateral filters define two types of filtering, *domain filtering* and *range filtering*. These filters rely on pixels being *close* to each other or pixels being *similar* to each other.

0×0	1×0	1	0	0
0×0	0×0	1	0	0
1	1	1	1	1
0	0	1	0	0
0	0	1	0	0

(a) The kernel applied to pixel $(0, 0)$

0×0	0×0	1×1	0	0
0×0	0×0	0×1	0	0
1	1	1	1	1
0	0	1	0	0
0	0	1	0	0

(b) The kernel applied to pixel $(0, 1)$

0	0	1	0	0
0	0	1	0	0
0×1	0×1	0×1	1	1
0×0	0×0	1×1	0	0
0×0	0×0	0×1	0	0

(c) The kernel applied to pixel $(4, 1)$

Figure 1.9: Various placements of the kernel in Figure 1.7 to the image in Figure 1.8a

Definition 6. Two pixels are *close* if they occupy a nearby spatial location.

Definition 7. Two pixels are *similar* if they have nearby values, possibly in a perceptually meaningful fashion.

Definition 8. *Domain Filtering* enforces closeness by weighing pixel values with coefficients that fall off with distance [37].

Definition 9. *Range Filtering* averages image values with weights that decay with dissimilarity. The weights depend on image intensity or color [37].

Bilateral filtering combines domain and range filtering in order to smooth the image while maintaining edges.

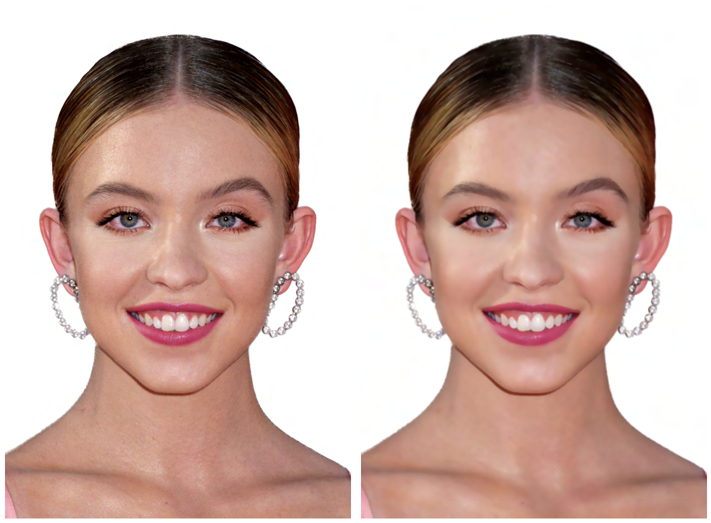

(a) Input image (b) Bilateral filtering

Figure 1.10: The result of bilateral filtering.

1.4.2 Sobel Operator

The Sobel operator performs a 2-D spatial gradient measurement to find regions of high spatial frequency (regions where image intensity changes rapidly) which correspond to edges [16]. The Sobel operator consists of two 3 x 3 kernels, as seen in Figure 1.11.

-1	0	1
-2	0	2
-1	0	1

(a) G_x

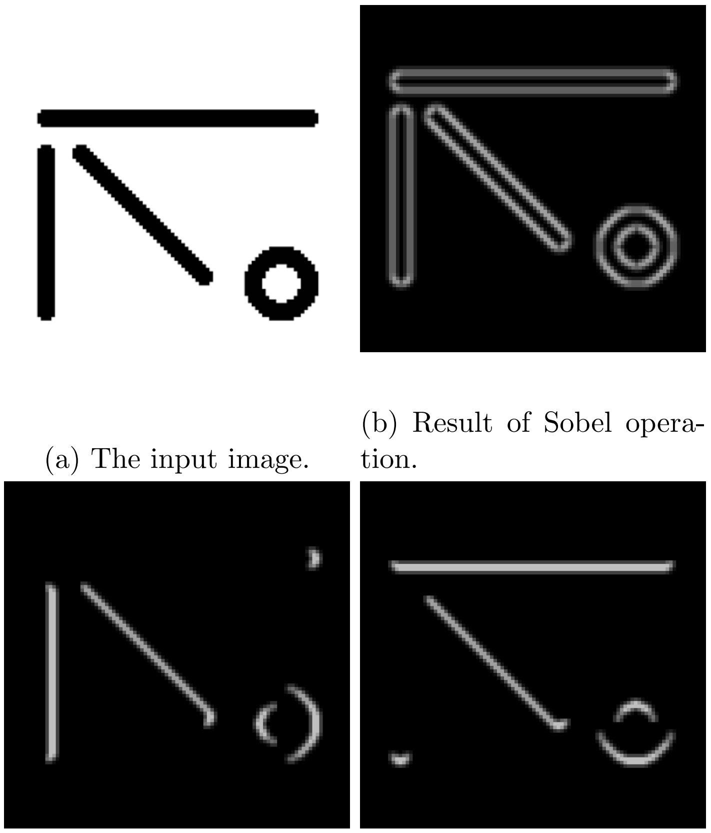
1	2	1
0	0	0
-1	-2	-1

(b) G_y

Figure 1.11: The Sobel convolution kernels

The orientation of these kernels allows them to respond maximally to edges that are vertical or horizontal relative to the pixel grid. Applying each of the kernels to the input image will result in a measurement of the gradient component in the horizontal and vertical directions, G_x and G_y. These gradients correspond to the rate of change in pixel intensity, areas with a greater rate of change indicate edges.

The absolute magnitude of the gradient at each pixel is given by $|G| = \sqrt{G_x^2 + G_y^2}$. The orientation of the edge, relative to the pixel grid is given by $\theta = \arctan(\frac{G_y}{G_x})$. If $\theta = 0$ then we have a perfectly horizontal edge where the maximum contrast from black to white runs from left to right [16]. Figure 1.12 shows the G_x and G_y components of the Sobel operator as well as the absolute gradient.


(a) The input image.

(b) Result of Sobel operation.

(c) The G_x component of the Sobel operation.

(d) The G_y component of the Sobel operation.

Figure 1.12: The Sobel operation and its components.

1.4.3 Canny Edge Detection

The Canny Edge Detection algorithm begins with noise reduction. Edge detection can become muddled if an image contains too much noise so Canny Edge Detection uses a 5×5 Gaussian Filter for noise reduction. Next the orientation of the edges in the input image are calculated. This is done using a Sobel operator as described in Section 1.4.2.

Once these values are found a full scan of the image is done to remove pixels that are not parts of the edges by checking if a pixel is a local maximum of its neighborhood in the direction of the gradient. The Canny Edge Detection algorithm defines two values `maxVal` and `minVal`. If a potential edge has intensity gradient that is greater than `maxVal` it is kept as an edge. If the intensity gradient is below `minVal` then it is discarded. If the intensity value is between `maxVal` and `minVal` the edge is kept if it is connected to pixels that have intensity gradients greater than `maxVal` and discarded if it has no such connections. Figure 1.13 shows edges detected using Canny edge detection.

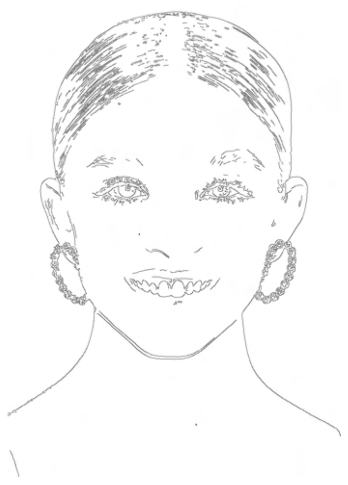


Figure 1.13: The edges returned by Canny edge detection.

1.4.4 Color Conversion

There are many ways to convert an image from one color space to another. In `OpenCV` the function `cvtColor` takes an input matrix, output matrix, and color conversion code. The `COLOR_RGB2GRAY` code transforms a color image to grayscale by averaging the R, G, and B components of the input image.

The `COLOR_RGB2LAB` code transforms a color image in the RGB color space to a color image in the $CIEL*a*b$ color space. See Appendix A for more detail.

1.4.5 Morphological Operations

Morphologies are a robust set of image processing operations that manipulate an input image based on shapes. The structuring element that is used to process the image can be of any size or shape and is often a rectangle, ellipse, or cross. The input image typically contains objects whose pixels will either be kept, removed, or expanded, based on the operation being performed. There are several types of morphological operations but they all follow this basic formula: we place the structuring element on each pixel of the input image and determine if there is a *fit*, *hit*, or *miss*, Figure 1.14. The pixel is kept (set to 1) or removed (set to 0) based on the specific operation.

Definition 10. A *fit* occurs when the structuring element covers only pixels that are part of the object in the input image [12].

Definition 11. A *hit* occurs when the structuring element covers both pixels that are part of the object and those that are not [12].

Definition 12. A *miss* occurs when the structuring element covers no pixels that are part of the object in the input image [12].

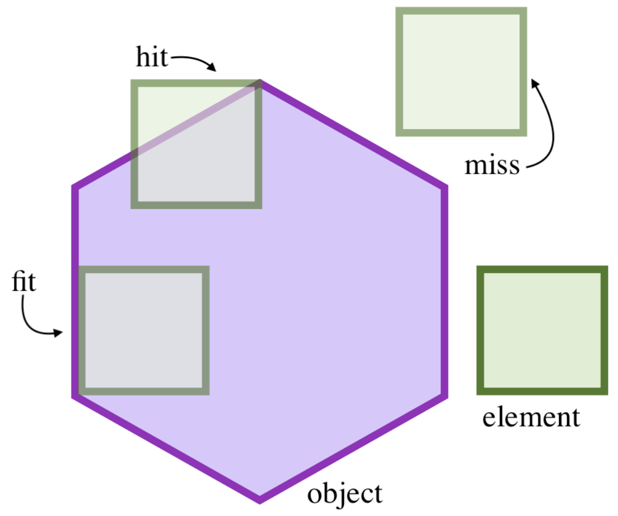


Figure 1.14: Morphology determinations.

There are four different morphological operations that uniquely use fits, hits, and misses to manipulate images.

1. **Erosion** removes pixels from the object's boundaries. If the structuring element is placed on a pixel the pixel is kept only if there is a fit, it is removed otherwise. Erosions are useful for smoothing lines or making them thinner, as shown in Figure 1.15.

2. **Dilation** expands the object's boundaries. If the structuring element is placed on a pixel and there is a fit or a hit the pixel is kept, it is removed otherwise. Dilations are useful for filling gaps in an image or thickening lines, as shown in Figure 1.16.

3. **Opening** is a compound operation, it is an erosion followed by a dilation. Openings are useful for removing noise from an image while maintaining the thickness of original objects and lines, as shown in Figure 1.17.

4. **Closing** is another compound operation, it is a dilation followed by an erosion. Closings are useful for removing small holes or black points in an object or image, as shown in Figure 1.18.

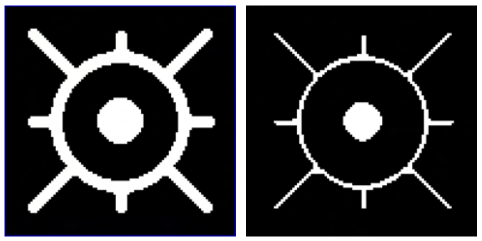


Figure 1.15: Morphological erosion by an elliptical element of size 4.

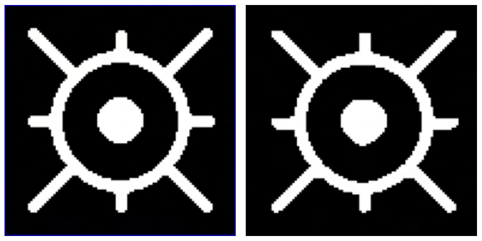


Figure 1.16: Morphological dilation by an elliptical element of size 4.

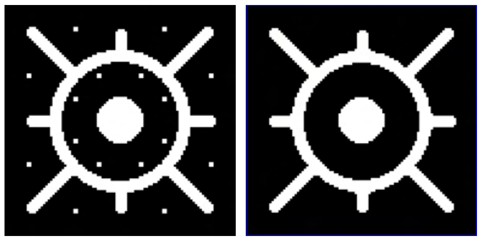


Figure 1.17: Morphological opening by an elliptical element of size 3 to remove noise.

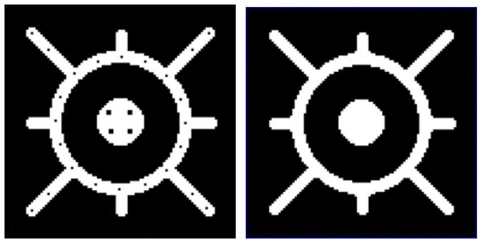
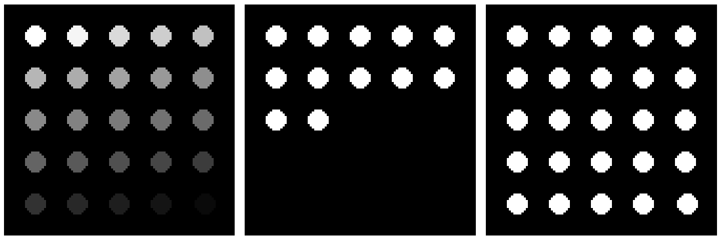


Figure 1.18: Morphological closing by an elliptical element of size 3 to fill holes.

1.4.6 Thresholding

In simple thresholding we set two values, `thresh` and `maxValue`. For each pixel we examine its value, if it is smaller than `thresh` then the pixel's value gets set to 0, otherwise it is set to `maxValue` [3]. Thresholding is frequently used for converting grayscale images to binary images.



(a) A gradient of grayscales (b) `maxValue` = 0 and `thresh` = 127 (c) `maxValue` = 0 and `thresh` = 255

Figure 1.19: A grayscale image thresholded at different values

Chapter 2

Extracting Feature Lines

The first step of any hedcut rendering is to extract the feature lines from an image [25, 36, 24]. The feature lines denote the outline of the face, eyes, nose, lips, etc. and also give an idea of the highlights of the face such as the cheek bones or the forehead. Accurately completing this step is critical for the later placement of dots. We found that sloppy detection of edges would compound issues in later steps.

2.1 Edge Detection

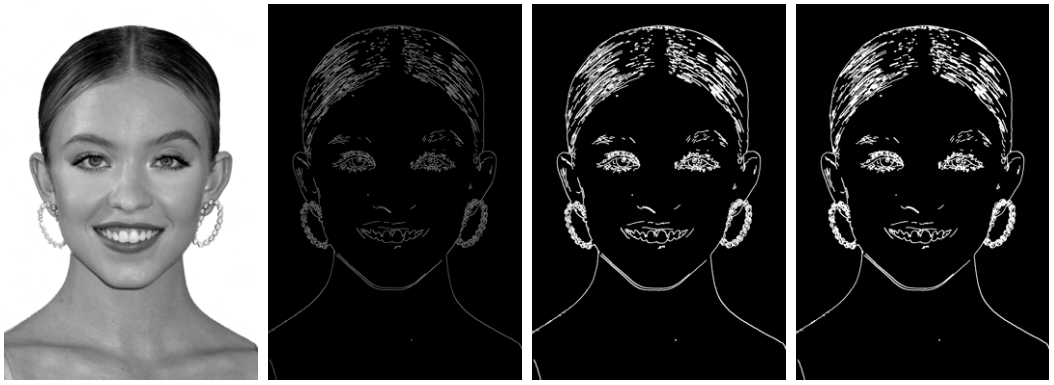
Sobel, Laplacian of Gaussian (LoG), and Canny are all viable edge detection algorithms. Kim et al. [25] opt to use a Difference of Gaussians (DoG) filter as an approximation of the LoG filter. In this work we found that the `OpenCV` Canny operator outperformed the DoG operator in terms of accuracy of feature extraction. We rely heavily on `OpenCV` operations here but attempt to combine them in novel ways to improve upon the results of Kim et al. [25].

First we convert the image to grayscale and apply the `blur` operator to remove noise from the input using a normalized box filter, see 2.1a. We apply the `canny` operator on the results to retrieve the initial edges, see Figure 2.1b. To refine these edges we apply a morphological dilation to close any small holes and thicken edges, see Figure 2.1c. Finally we apply a median filter to smooth the edges, see Figure 2.1d. Figure 2.2 shows these refinements in more detail.

2.1.1 Thresholding Edges

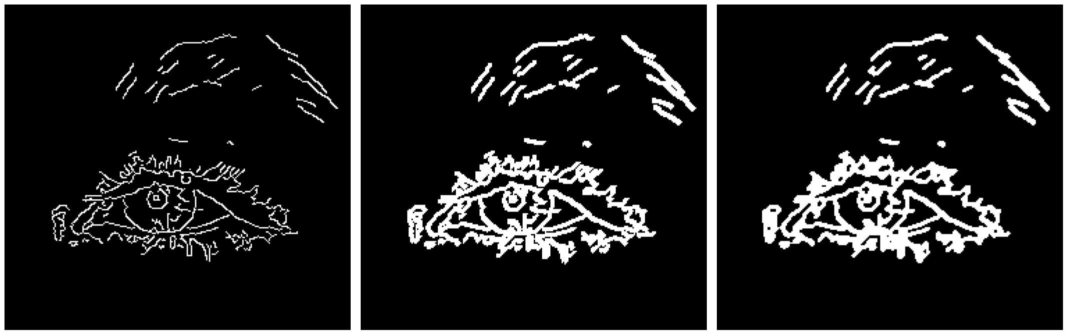
After the edges have been found we threshold the image to remove any short edges as done in Kim et al. [25]. For this we use an `OpenCV` operator[1] which takes an input and binary image and finds all connected components in that image. This function returns two output matrices, `stats` and `labels`, which are used to find short edges. In the `labels` output matrix each connected component is colored based on its index i.e. the *nth* component has a grayscale value of *n*.

[1]`OpenCV connectedComponentsWithStats`



(a) After grayscale conversion and blurring. (b) Initial edges detected with the Canny operator. (c) Edges thickened with morphological dilation. (d) Edges smoothed with median blur filter.

Figure 2.1: Edges detected before thresholding.



(a) Edges detected after canny filter. (b) Edges after morphological dilation. (c) Edges smoothed with median blur.

Figure 2.2: Closeup of edge refinement process.

To determine if the *ith* connected component meets the threshold value we must isolate the sub-matrix that contains that individual edge. This sub-matrix may, however, contain other partial edges. These can easily be removed by analyzing the color of each pixel and setting all pixels without a value of i to black. Next we can estimate the length of the *ith* component by extracting its morphological skeleton. Skeletonization of a binary image is the process of reducing all curvilinear objects to a line that is no more than two pixels wide, like the letters in Figure 2.4 [11, p. 474]. The line that remains can be used as a rough estimation of the object's length, as shown in Figure 2.3. Our work uses the algorithm for extracting a morphological skeleton described in a blog post by Félix Abecassis [5]. The skeleton can be extracted using the morphological operations described in Section 1.4.5. First a morphological opening is performed and the result is inverted. Then a bitwise `or` of the opening and the original image is computed. We repeat this process and stop before the final iteration that would result in a completely empty image. We use the number of white pixels in the skeleton as an estimate for the edge's length.

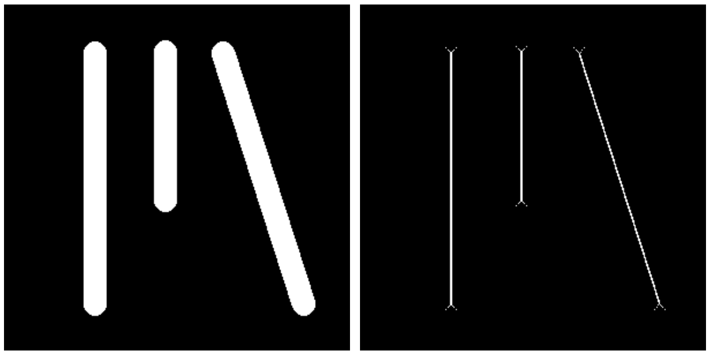
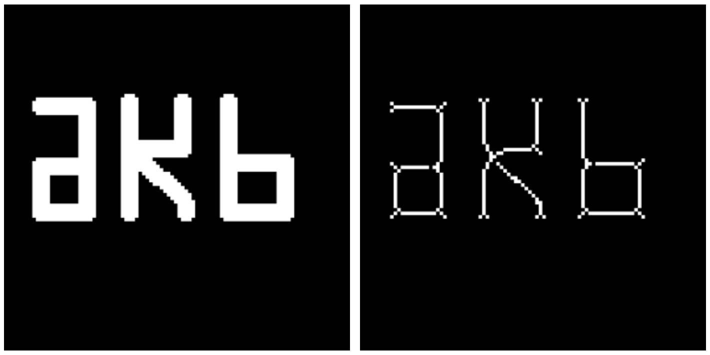


Figure 2.3: The skeleton of lines as an estimation of their length.



(a) My initials. (b) The skeleton of 2.4a.

Figure 2.4: An image and its skeleton.

We use a dictionary, `remove`, to indicate whether or not a component, i, meets the threshold by setting

```
remove[i] = !(meetsThreshold(skel, threshold))
```

where i is the edge's color in the `labels` matrix and `meetsThreshold(skel, threshold)` returns `true` if the number of white pixels in the skeleton is greater than or equal to the `threshold` parameter.

Finally, we can loop over the `labels` matrix and for each pixel we extract its color c. If `remove[c]` is true we set that pixel to have color 0, otherwise the color is 255. The complete pseudocode for the thresholding algorithm can be found in Appendix B.1. Figure 2.5 shows the result of thresholding to remove edges with a length less than 50. After thresholding we apply a median filter one more time to smooth edges, see Figure 2.6.

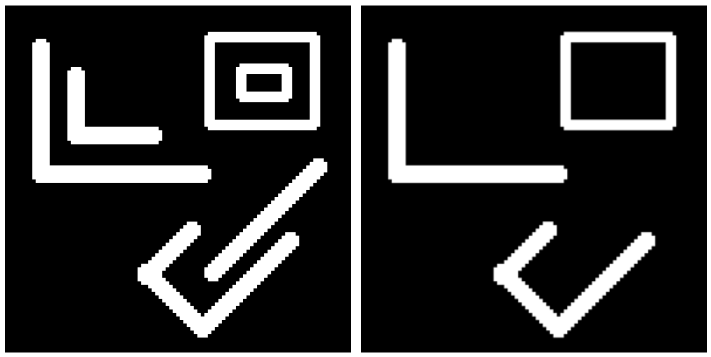
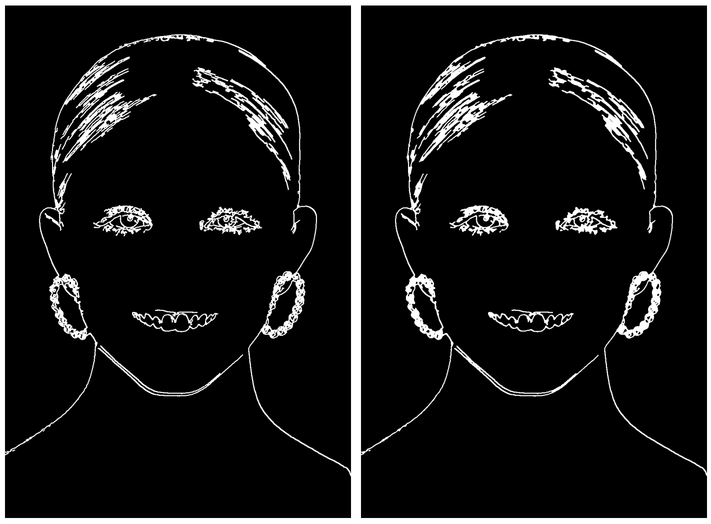


Figure 2.5: Thresholding to remove edges of length less than 50 px.



(a) Thresholded edges from Figure 2.1d.

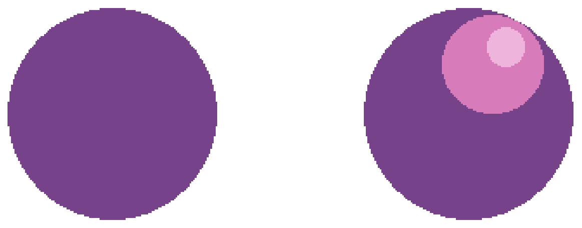
(b) Edges after final median blur.

Figure 2.6: The final edges detected with our algorithm.

2.2 Isophote Detection

A major difference between Kim et al. [25] and other approaches to hedcut creation is the consideration of *isophotes*. Isophotes are useful for isolating and analyzing local shapes or for representing depth relationships [25]. An artist might add depth to a sphere by adding isophotes to capture shading and highlights. In Figure 2.7a the sphere appears flat. In 2.7b the lighter regions help give an illusion of depth. In a face isophotes bound regions of different depths such as the apples of the cheek, forehead, tip of the nose, etc[2].

Definition 13. An *isophote* is a curve connecting points of equal intensity.



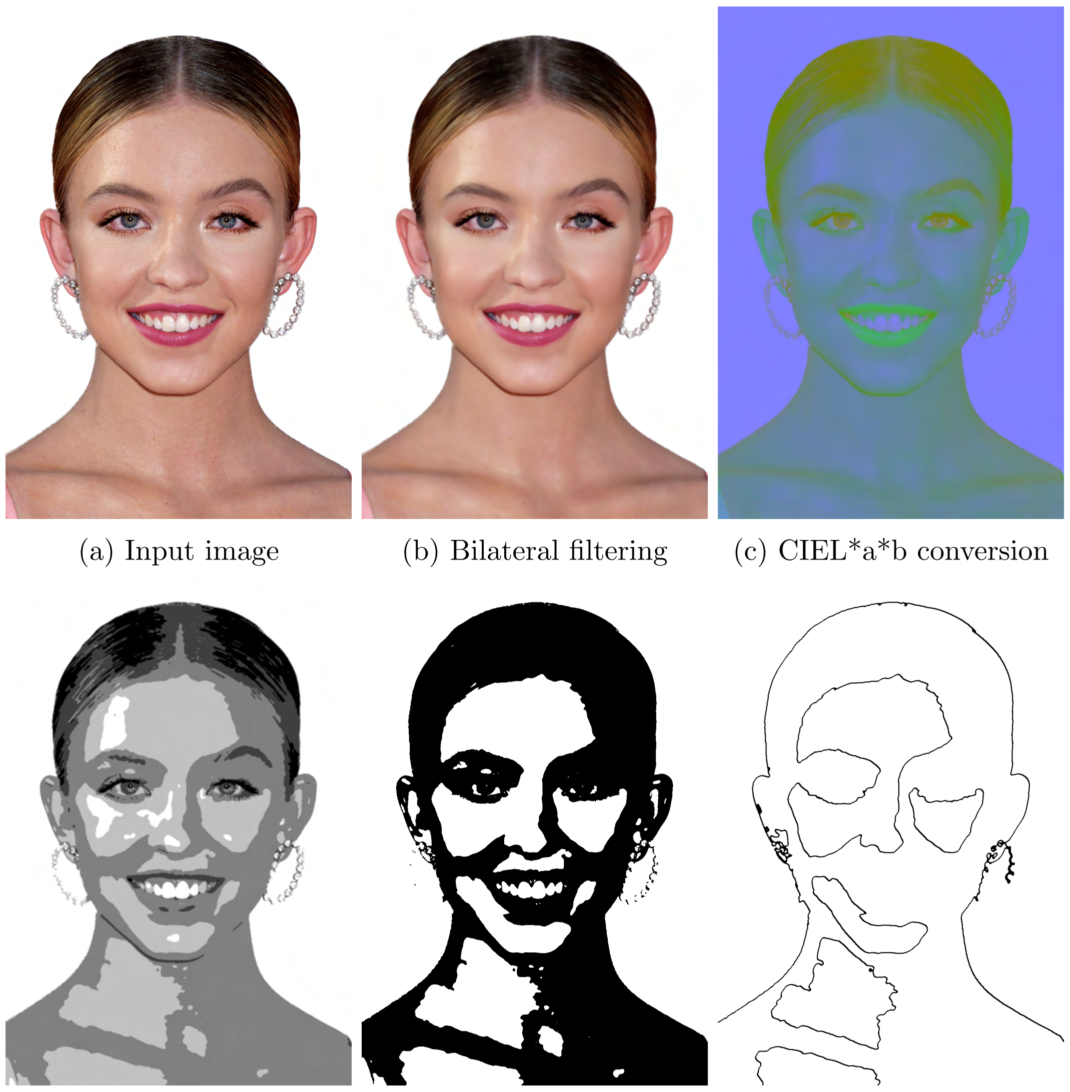
(a) A flat sphere. (b) A sphere with depth.

Figure 2.7: In 2.7b isophotes are added to give the illusion of depth

It is easy for humans to detect isophotes in a face. Even if every pixel in Sydney Sweeny's cheeks in Figure 2.8a are not the exact same intensity we can still tell that this region is a highlight and has a different depth than her chin. A computer, however, would have trouble detecting these isophotes because there are few areas where pixels have the exact same intensity. To detect and isolate isophotes we must quantize our input image. Quantizing is the process of grouping or "binning" intervals of data into a single value or quantum. First a bilateral filter is applied to smooth the image, see Figure 2.8b. Next, we convert the image from RGB space to $CIEL*a*b*$ space, see Figure 2.8c. Recall that in $CIEL*a*b*$ space the L value of a pixel gives its luminosity. Then we convert the image to grayscale by retaining only the L component of each pixel and quantize to reduce the ranges of gray, see Figure 2.8d. This effect is also referred to as posterization.

Posterization expands the areas where pixels have the exact same value. We can use the segmented areas of the posterized image to find isophotes. First, for some predefined threshold value t, we select only the top t segmented areas by setting the pixels in these areas to 255 and the pixels in areas that are too dark to 0, see Figure 2.8e. Then, we perform edge detection on these segments. The result is the isophotes of the face, see Figure 2.8f.

[2]You can also think of the isophotes of the face as the areas where one would place highlighter or contour when doing makeup. The purpose of these products is, after all, to shape the face by adding lines of constant intensity which give the illusion of depth



(a) Input image (b) Bilateral filtering (c) CIEL*a*b conversion

(d) Quantizing the L value (e) Thresholding (f) Edge detection

Figure 2.8: The intermediate steps for isophote detection.

Chapter 3

Stippling

Once we have extracted the edges and isophotes of an image we use these lines to place dots. First we generate a weighted distance map, see Figure 3.1a. Then we produce an offset map by evenly spacing lines throughout the weighted distance map, see Figure 3.1b. The offset map is used to place the initial stipple dots which are then adjusted so that they follow the flow of the initial feature lines, see Figure 3.1c and Figure 3.1d. Finally the stipple dots are rendered with dot size being informed by the luminosity in the initial input image, see Figure 3.1e.

3.1 Weighted Distance Maps

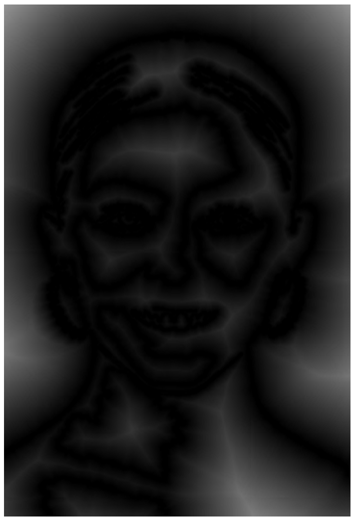
We want to place stipple dots evenly throughout our rendering and we want these dots to follow the flow of the feature lines. To accomplish this we must know the distance between any given pixel and the feature line that it is closest to. This is done by generating a distance map which is a grayscale image where each pixel's value gives the distance between that pixel and the nearest *seed* pixel in the input image. For this work the input images were the edges and isophotes detected in Sections 2.1 and 2.2, respectively.

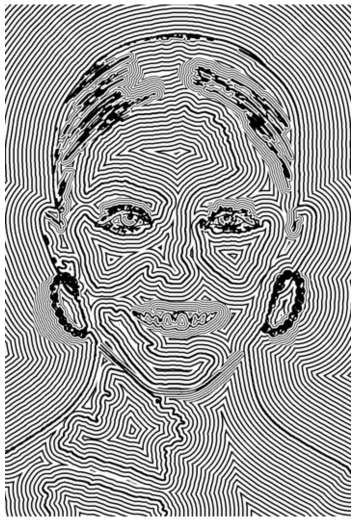
Definition 14. In a binary image with a white background a *seed pixel* is a pixel that has value 0 (black), the black pixels in Figure 3.1c.

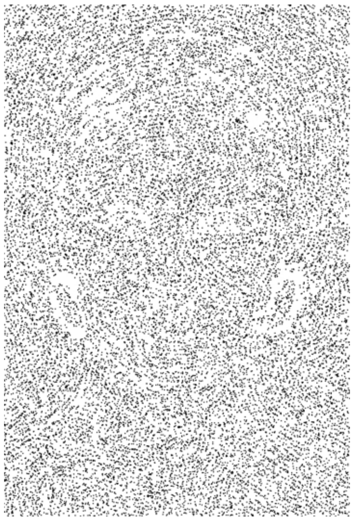
Figure 3.2 shows a distance map for the simple input image in Figure 3.2a. Note that the final distances are rounded integers. For a pixel, p, that makes up a feature line, such as pixel $(2, 0)$, p is given a value of 0 in the distance map since the distance between p and the nearest seed pixel is the distance from p to itself.

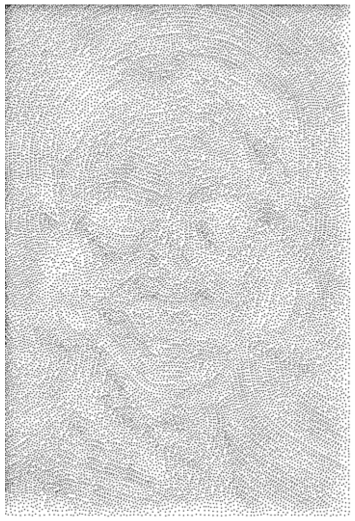
In a weighted distance map certain seed pixels, say those that make up isophotes, are given priority and then the map is skewed in that direction. A weighted and unweighted distance map for an image with edges and isophotes is shown in Figure 3.3. Section 3.1.2 covers the generation of these maps in detail.

The first step of generating these maps is to find the distance of each pixel from a feature line. Initially I approached this task with the Jump Flood Algorithm which is done in Kim et al. [25]. I did not end up using this algorithm in my final approach because it was noticeably slow for large images. Instead I opted to use Felzenszwalb and


(a) Weighted distance map.


(b) Offset map.


(c) Initial dots†.


(d) Adjusted dots†.

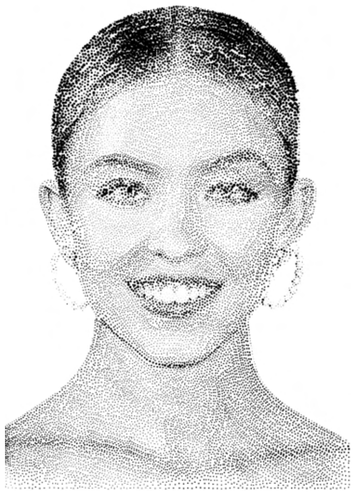
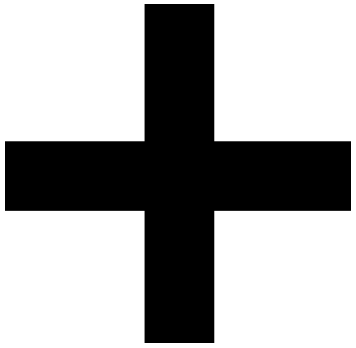

(e) Rendered dots.

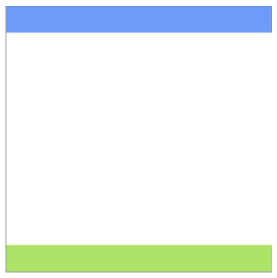
Figure 3.1: The intermediate steps for stippling.


(a) A very simple image.

$$
\begin{array}{l}
[\quad [2,\ 1,\ 0,\ 1,\ 2] \\
\quad\ \ [1,\ 1,\ 0,\ 1,\ 1] \\
\quad\ \ [0,\ 0,\ 0,\ 0,\ 0] \\
\quad\ \ [1,\ 1,\ 0,\ 1,\ 1] \\
\quad\ \ [2,\ 1,\ 0,\ 1,\ 2] \quad]
\end{array}
$$

(b) The corresponding weighted distance map represented numerically.

Figure 3.2: An image and its distance map.


(a) A simple image with an edge (blue) and isophote (green).

```
[[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
 [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
 [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
 [ 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 ]
 [ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
 [ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
 [ 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 ]
 [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
 [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]]
```

(b) The corresponding unweighted distance map represented numerically.

```
[[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
 [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
 [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
 [ 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 ]
 [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
 [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
 [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
 [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]]
```

(c) The corresponding weighted distance map represented numerically[a].

[a]Recall that pixels only have whole integer values. The pixels in row 9, for example, should have a value of 0.5 but are rounded down to 0. See Section 3.1.2 for more detail.

Figure 3.3: An image with an edge and an isophote and its weighted and unweighted distance maps.

Huttenlocher's Sampled Distance Transformation as described in [15] which performs much better[1].

3.1.1 Sampled Distance Transformation

The Felzenszwalb and Huttenlocher method utilizes a generalized distance transform for arbitrary functions on a grid. This allows for distance transforms on real-valued images rather than exclusively binary valued distances. Consider the regular grid $\mathcal{G}$ and a function on the grid $f : \mathcal{G} \to R$. Felzenszwalb and Huttenlocher refer to a function on a grid as a *sampled function.* Given a sampled function, f, a function for calculating distance, d, the distance transform of f is then the function $\mathcal{D}_f : \mathcal{G} \to \mathbb{R}$ such that

$$\mathcal{D}_f(p) = \min_{q \in \mathcal{G}}(d(p,q) + f(q))$$

In other words, given a point p, we are finding the point q such that the distance between p and q, $d(p,q)$, is small and the function $f(q)$ is also small.

Felzenszwalb and Huttenlocher method first computes the distance map in one dimension only. Then the same algorithm is run in the second dimension to compute the final distance in a two dimensional space. This process can be repeated for arbitrary dimensions.

One Dimension

Let $\mathcal{G} = \{0, \cdots, n-1\}$ be a one-dimensional grid, let $f : \mathcal{G} \to \mathbb{R}$ be a function on the grid. We can compute the squared Euclidean distance transform by using $d(p,q) = (p-q)^2$. So the distance transform for f is:

$$\mathcal{D}_f(p) = \min_{q \in \mathcal{G}}((p-q)^2 + f(q))$$

Felzenszwalb and Huttenlocher note that for each point $q \in \mathcal{G}$ the distance transform of f be bounded by a parabola rooted at $(q, f(q))$, we will refer to this parabola as "the parabola at q". Note that the parabola rooted at $(q, 0)$ will give the distance between q and any other point $s \in \mathcal{G}$. To take the function f into account we must shift the parabola rooted at $(q, 0)$ vertically by $f(q)$. Therefore the distance between q and s will lie on the parabola rooted at $(q, f(q))$.

We can define the distance transform by the lower envelope of these parabolas as shown in Figure 3.4. For any point $p \in \mathcal{G}$ we can find $\mathcal{D}_f(p)$ by computing the height of the lower envelope at p. To compute $\mathcal{D}_f(0)$ we note that the lower envelope at $p = 1$ is given by the parabola rooted at 0. So $\mathcal{D}_f(0)$ is the height of that parabola at the horizontal position of 1. To compute $\mathcal{D}_f(1)$ we note that the lower envelope at $p = 1$ is also given by the parabola rooted at 0. So $\mathcal{D}_f(1)$ is the height of that parabola at the horizontal position of 1.

[1]The Jump Flood Algorithm has $O(N \log N)$ running time whereas Felzenszwalb and Huttenlocher's method has $O(N)$ running time.

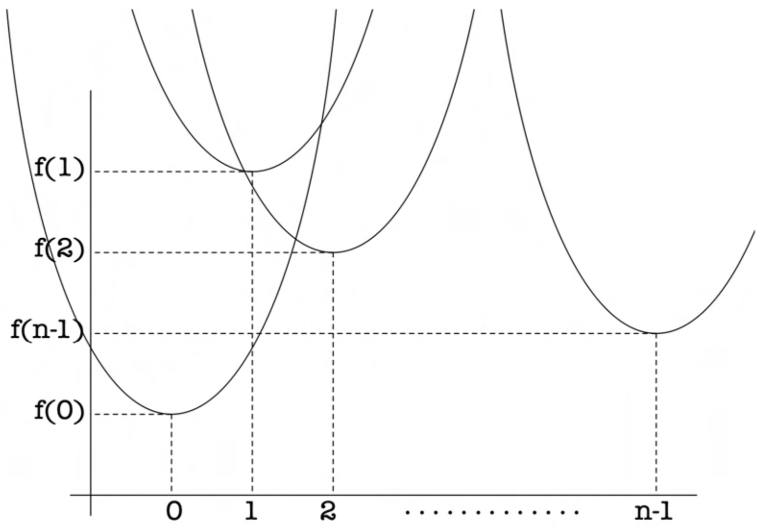


Figure 3.4: The distance transform as the lower envelope of n parabolas from Felzenszwalb and Huttenloche, 2012. Image reprinted under CC BY 3.0.

To compute this lower envelope we can note that any two parabolas in the distance transform intersect at exactly one point. The intersection of the parabolas with rooted at $(q, f(q))$ and $(r, f(r))$ is given by:

$$s = ((f(r) + r^2) - f(q) + q^2)/(2r - 2q)$$

The structure of the lower envelope is tracked using two arrays, v and k. The array v gives the horizontal position of the root of the i-th parabola. In Figure 3.4 $v[0] = 0$ and $v[1] = 2$. The parabola rooted at $(1, f(1))$ is not part of the lower envelope so there is no i such that $v[i] = 1$. The z array tracks the range in which the i-th parabola of the lower envelope is below the others. $z[i]$ gives the start of that range and $z[i+1]$ gives the end of that range. In Figure 3.4 $z[0] = -\infty$, $z[1] \approx 1.5$, and $z[n] = \infty$. We also keep a variable k to track the total number of parabolas in the lower envelope.

Parabolas are added or removed from the lower envelope as follows. Consider the parabola at q which is not yet determined to be in the lower envelope. Let s be the intersection between the parabola at q and the parabola at $v[k]$ (the rightmost parabola of the lower envelope). $z[k]$ will give the start of the range in which the parabola at $v[k]$ is below all others in the lower envelope and $z[k+1]$ will be ∞. If $s > z[k]$ then the parabola at q and the parabola at $v[k]$ intersect after the point at which the parabola at $v[k]$ is below the others. This means the parabola at q will be below the rightmost parabola in the range $z[k]$ to ∞. In this case, the parabola at q must be added to the lower envelope. If $s < z[k]$ then the parabola at $v[k]$ intersects with the parabola at $v[k]$ parabola before the parabola at $v[k]$ is below the others. This means the parabola at q will be below the others before the parabola at $v[k]$ becomes below the others. In this case the rightmost parabola should be removed from the lower envelope and the parabola at q should be added to the lower envelope.

Figure 3.5 demonstrates these cases.

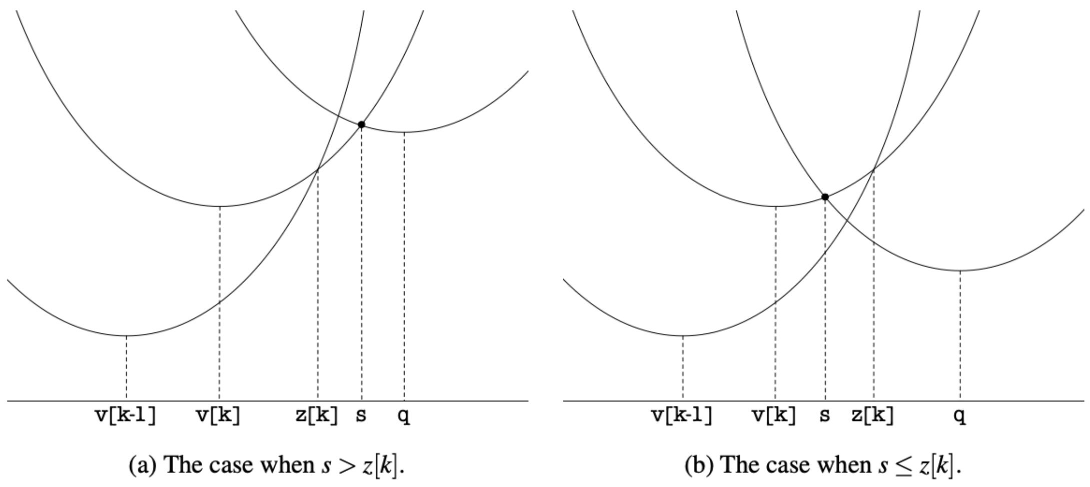

v[k-1]
v[k]
z[k]
s
q
(a) The case when $s > z[k]$.
v[k-1]
v[k]
s
z[k]
q
(b) The case when $s \leq z[k]$.

Figure 3.5: The two cases when adding the parabola from q to the lower envelope from Felzenszwalb and Huttenlocher, 2012. Image reprinted under CC BY 3.0.

The second step of this algorithm populates the values of $\mathcal{D}_f(p)$ by determining the height of the lower envelope at that grid position p. A proof of correctness for these algorithms is given in Felzenszwalb and Huttenlocher's paper [15].

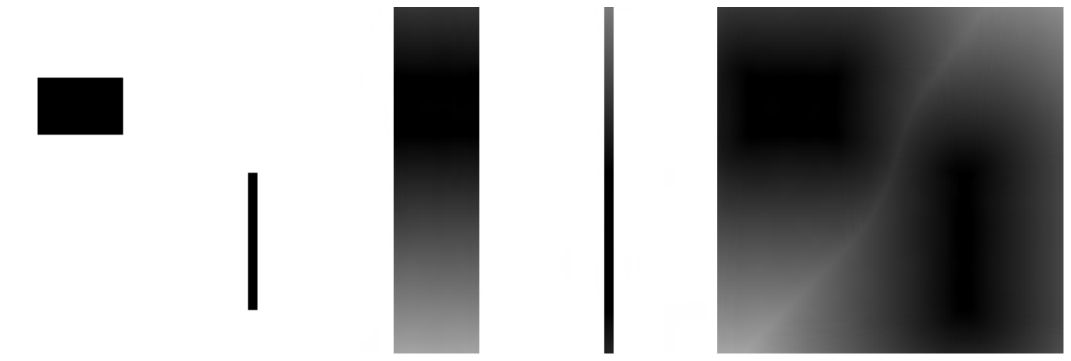
Two Dimensions

Now consider a two-dimensional grid $\mathcal{G} = \{0, \cdots, n-1\} \times \{0, \cdots, m-1\}$ and the function $f : \mathcal{G} \times \mathbb{R}$. The two-dimensional distance transform of f under the squared Euclidean distance is given by

$$\begin{aligned}\mathcal{D}_f(x, y) &= \min_{x', y'}((x - x')^2 + (y - y') + f(x', y') \\ &= \min_{x'}((x - x')^2 + \min_{y'}((y - y')^2 + f(x', y'))) \\ &= \min_{x'}((x - x') + \mathcal{D}_{f|_{x'}}(y))\end{aligned}$$

$\mathcal{D}_{f|_{x'}}(y)$ is a one dimensional distance transform with f restricted to column x'. Thus to compute a two-dimensional distance transform the distance transform is first computed in one dimension along each column of the grid and the result of the distance transform is stored in that grid. Then the distance transform is computed in one dimension along each row of the grid.

In the resulting grid, however, a pixel, p, will not hold the Euclidean distance from p to the closest seed pixel. Instead, p will hold the squared Euclidean distance between the two pixels. One final pass must be done over the grid to take the square root of each pixel value. The two passes of the distance transform are shown in Figure 3.6. Appendix B.2 gives the pseudocode for the full algorithm.



(a) An input image for distance transformation. (b) Transform of input along each column. (c) Transform of 3.6b along each row.

Figure 3.6: An image and its distance map.

3.1.2 Applying Weight / Priority

Creating a weighted distance map is simple. First a distance map is created for the edges and isophotes. Figure 3.7 shows the distance maps created using the edges and isophotes generated in Sections 2.1 and 2.2. The edges and isophotes are each given a priority or weight, w_e and w_i, respectively. For every pixel p in the input image let p_e and p_i be the values that correspond to p in the distance map for the edges and the distance map for the isophotes, respectively. The value of p in the weighted distance map is computed as $D(p) = \min(\frac{p_e}{w_e}, \frac{p_i}{w_i})$. For my work, edges were given a weight of 1 and isophotes were given a weight of 2. This gives isophotes a higher priority which enhances the perceived depth. Figure 3.8 shows how different weights affect a final distance map. In Figure 3.8d we can see that giving isophotes a weight of 5 and edges a weight of 1 causes the map to be very skewed in the direction of the isophotes since the pixels near to the isophotes are very dark meaning their values are small.

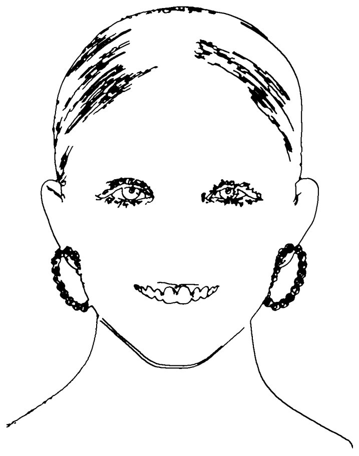
3.2 Offset Maps

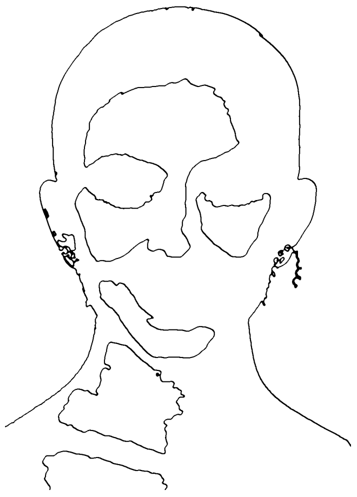
Once we have a weighted distance map we generate an offset map following the method used in Kim et al. [25]. An offset map consists of evenly spaced lines that follow the edges and isophotes of an input image. The lines are given a width w_0 and are spaced at an interval of $(l - w_0)$. Following Kim et al.[25] we gave w_0 a value of 1 and l a value of 6, however we later allowed for l to be specified by the user. Figure 3.9 shows how the l value affects the offset map.

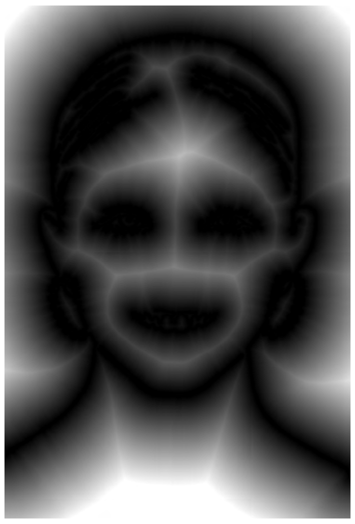
For every pixel, p, we can determine if p will make up an offset line by computing the distance between p and the nearest feature line. If p has a distance d from the nearest feature line (calculated using the weighted distance map) we can compute the distance Δ between p and the previous offset lane as:

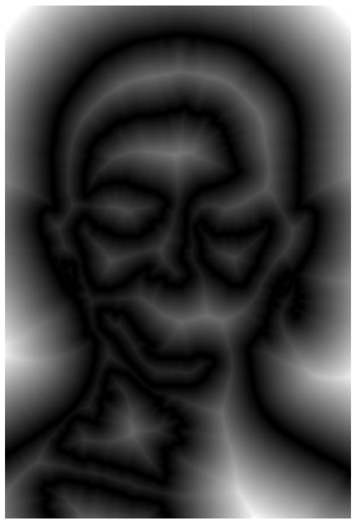
$$\Delta = \lceil d/l \rceil \cdot l - d$$

If $\Delta \leq w_0$ then p should be part of an offset line and it is given a value of 0.


(a) Input edges.


(b) Input isophotes.


(c) The distance transform of Figure 3.7a.


(d) The distance transform of Figure 3.7b.

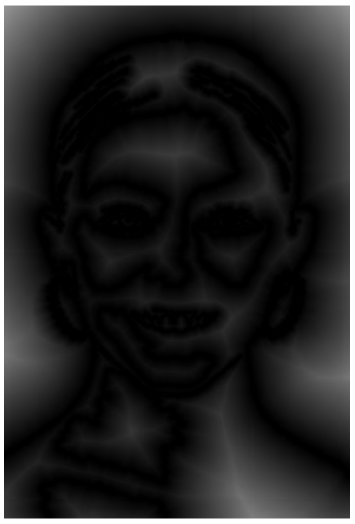
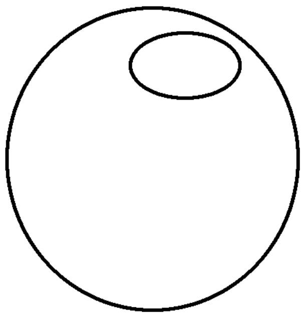
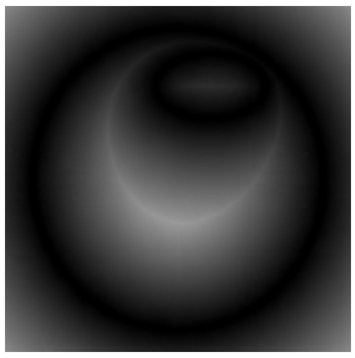

(e) Weighted distance transform.

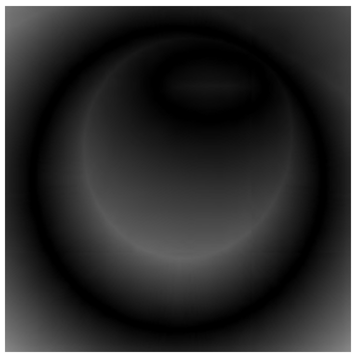
Figure 3.7: The edges and isophotes extracted from Figure 6 and the resulting distance maps.



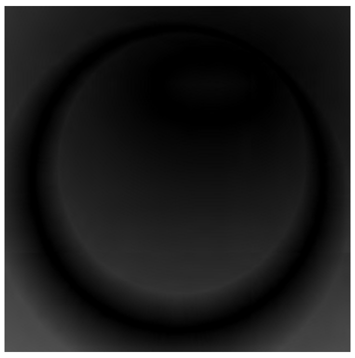
(a) A sphere with an isophote.



(b) The weighted distance transform with edge weight of 1 and isophote weight of 1.



(c) The weighted distance transform with edge weight of 1 and isophote weight of 2.



(d) The weighted distance transform with edge weight of 1 and isophote weight of 5.

Figure 3.8: An image and various weighted distance maps.

Otherwise p resides in an offset lane between offset lines. In this case p is given an *id* to designate which offset lane p is a part of. This is done by setting p's value to $\lceil \frac{d}{l} \rceil$. We also generate a strictly visual offset map for debugging purposes. In this map the offset lines are given a value of 0 and each segment between is given a value of 255.

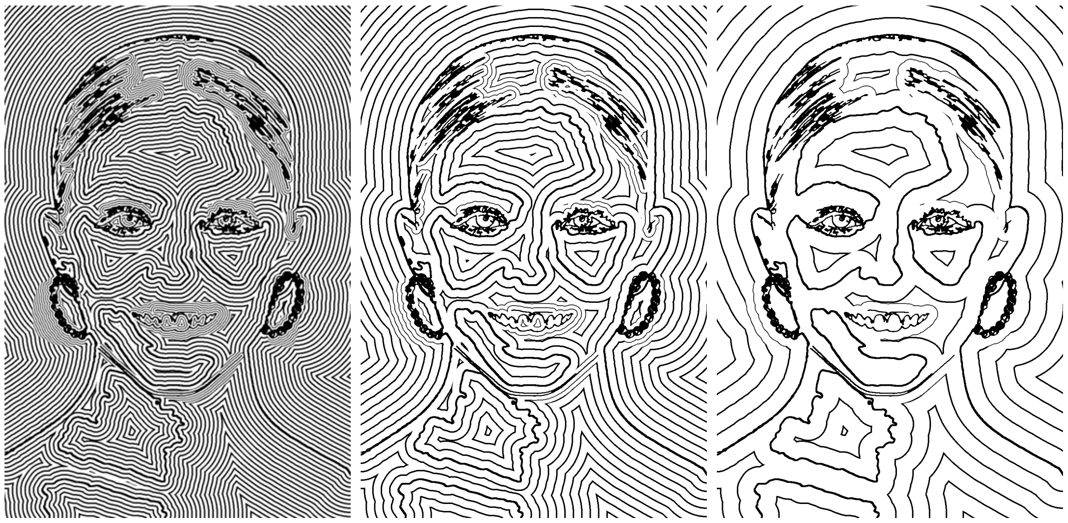

(a) Offset map with $l = 6$. (b) Offset map with $l = 12$. (c) Offset map with $l = 24$.

Figure 3.9: Offset maps generate from Figure 3.7e with different l values

3.3 Placing Dots

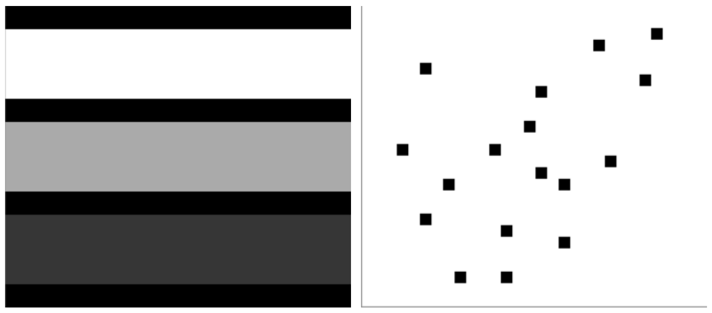
Given the offset and feature lines we are now ready to place and render the stipple dots. To do so we follow the Constrained Lloyd relaxation method in Kim et al. [24]. First we randomly scatter seeds throughout the image. Then we iteratively adjust each seed so it aligns with the center of an offset lane to maintain feature flow. Once the seed placement is finalized we render the final dots, adjusting the size based on the luminosity of the input photograph.

3.3.1 Initial Dot Placement

Given an offset map we want to randomly scatter dots throughout the image while avoiding offset lines. To do so we pass over the pixels of an offset map, if the pixel is part of an offset line we ignore it. Otherwise we generate a random number $r \in [0, 1]$ and place a seed pixel at this location if $r \leq \frac{1}{d^2}$ where d is the distance between the offset lines. Figure 3.10b shows the initial seeds generated from Figure 3.10a.

3.3.2 Adjusting Dots

Adjusting the dots requires creation of a Voronoi diagram. A Voronoi diagram is a data structure that divides a plane according to the nearest neighbor rule [10]. Let S



(a) A simple offset map. (b) The initial seed pixels.

Figure 3.10: A simple offset map and the seeds it generates.

be the set of seed pixels in the input image. For each seed pixel in the input image, $s \in S$, we find the set of pixels P_s such that:

$$d(s,p) < d(s',p) \; \forall \; s' \in S \setminus \{s\}$$

where $d(s,p)$ is the Euclidean distance between s and p. That is, given a set of seed pixels in a plane a Voronoi diagram will divide the plane into sections, Voronoi cells, such that all pixels in a given section are closer to the seed pixel in that section than they are to any other seed pixel in the plane. In Figure 3.11 all the blue pixels in the upper left corner are closer to the black pixel in that cell than any other black pixel in the image.

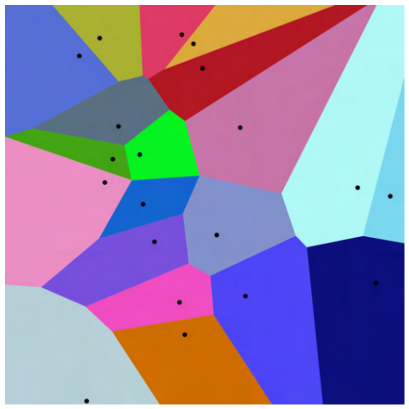


Figure 3.11: Voronoi diagram from Wikimedia Commons. Image reprinted under CC BY-SA 4.0.

Once the initial dots are scattered throughout the image they are adjusted using Lloyd relaxation as described in Kim et al. [24]. We iteratively construct a Voronoi diagram from our seed pixels and adjust the seed pixels to be the centroids of the Voronoi cells. In Kim et al. [24] the Voronoi diagram is generated using the Jump Flood algorithm. In our approach we modified the Sampled Distance Transform described in Section 3.1.1 so that each pixel records its seed pixel in addition to its distance from said seed pixel. This modification is simple and does not affect the time of complexity of the algorithm.

To update the position of a seed pixel $s \in S$ we compute the weighted average of the pixels in the Voronoi cell containing s as:

$$c = \rho^{-1} \sum_i w_i \cdot x_i$$

where x_i is the i-th pixel in the cell, w_i is the associated weight, and $\rho = \sum_i w_i$ is the weight normalization term. This process moves the seed pixels towards the centers of the offset lanes.

In the Kim et al. [24] approach the offset lines are used to constrain the Voronoi cells so they eventually align with the offset lines. In the above formula if x_i is part of an offset line it is removed by setting $w_i = 0$. If a Voronoi cell is divided by an offset line there will be two non-offset line sections. Among these sections we consider only pixels that are in the same section as the seed pixel. Recall in Section 3.2 we tagged each non-offset line pixel with an *id* that corresponded to the section between offset lines it belonged to. We use each pixel's *id* to ensure that a pixel is in the same section as its seed. If it is we assign it weight $w_i = 1$, otherwise $w_i = 0$. This method moves the centroids of the Voronoi cells towards the center of the offset lanes and ensures that centroids do not move between lanes.

Kim et al. [24] prevent the dots from looking too structural by weakening the offset line constraints in areas far from feature lines. Instead of setting $w_i = 0$ for all offset lines they are given a weight proportional to their distance values:

$$w_i = min\{D(x_i)/D_w, 1\}$$

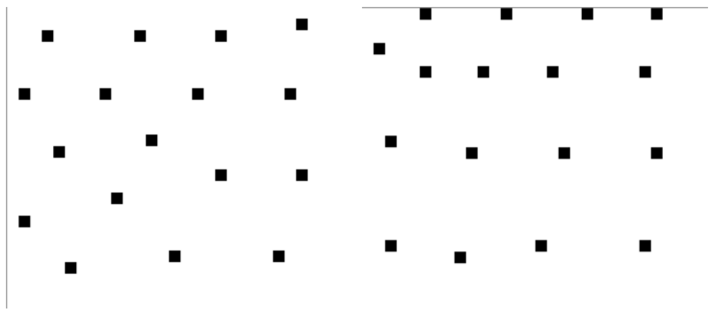
where D_w is the distance at which the offset line constraints will no longer have effect. Kim et al. [24] set $D_w = 100$.

The Lloyd algorithm is iterated t_1 times without offset line constraints to spread the initial set of dots. Then the algorithm is iterated t_2 times alternating the Lloyd algorithm with and without the offset line constraints to avoid clustering while spreading the dots more evenly. Kim et al. [24] set $t_1 = 10$ and $t_2 = 30$. Figure 3.12 shows the seed pixels from Figure 3.10b after adjustment with offset line constraints and after adjustment without offset line constraints. Notice how in Figure 3.12a the seeds clearly avoid the offset lines, but they are less centered within offset lanes. In Figure 3.12b the dots are drawn closer to the offset lines but are more strictly centered in the offset lanes.

3.4 Rendering Dots

Once the dots are adjusted the final image can be rendered. To improve image quality we follow Kim et al. [24] and render an image that is six times bigger than the input. The dot size at a pixel x is inversely proportional to $\mathcal{T}(x)$ where $\mathcal{T}$ is the normalized grayscale version of the input image. The dot size is calculated as:

$$s(x) = s_{max} \cdot (1 - \mathcal{T}(x))^{\gamma}$$



(a) Figure 3.10b after 10 iterations without offset line constraints.

(b) Figure 3.12a after 30 iterations with offset line constraints.

Figure 3.12: A simple offset map and the seeds it generates.

where s_{max} is the maximum possible dot size. We found that s_{max} strongly impacted the quality of the final image so we allowed this parameter to be specified by the user as described in Section 4.2. Kim et al. [24] use γ to incorporate gamma correction for tone control and set $\gamma = 1.2$ by default.

Chapter 4

Algorithmic Assessment

4.1 Evaluating NPR Renderings

As stated in Rosin et al. [34] “progress in science is best served when there are means to evaluate and compare theories and method”. In the field of NPR, however, it is difficult to compute a quantitative score that reflects the aesthetic qualities of the rendered output so we must fall back to subjective evaluation. Isenberg [20] points out that the seemingly simple question of “comparing handmade images to NPR images” is in fact rather complicated since there is no well defined notion of what it means to “compare” the two styles.

We could, for example, compare how well NPR programs perform on a variety of input images, as opposed to hand-drawn renderings of similar inputs. Rosin et al. [34] present a method for evaluating NPR techniques that uses standardized image sets for benchmarking. They organize a diverse collection of portraits into levels of difficulty. These sets include variation in face types, posing, emotion, accessories, as well as range in the conditions and environments in which people are photographed. For our work we compiled two sets of input images that correspond to level 1 and level 2 difficulty as described in Rosin et al. [34]. These sets and the characteristics that define them are found in Appendices C.1 and C.2

We could also compare NPR and hand-drawn images by asking how easy it is to distinguish the two styles. Isenberg mentions a NPR “Turing test” which asks people to distinguish a handmade rendering from an NPR one as an effective method for comparison. Researchers have also sought to compare the two styles through statistical approaches. Maciejewski et al. [26], for example, employ a statistical texture analysis on the distribution of stipple dots by examining the gray scale textures in NPR and hand-drawn images. They found that even NPR stippling techniques that incorporate randomness can still be easily distinguished from hand-drawn images due to other regularities. This type of analysis, however, is out of the scope of this thesis. To assess the quality of our results we examined how accurately our renderings included the “core qualities of hedcuts” as outlined in Section 0.1.1.

4.2 Initial Results

After completing all steps in [25] we were able to get results that resembled a stipple drawing as seen in Figure 4.1. If we compare this rendering to the qualities outlined in Section 0.1.1 we can see that it...

- ✓ are small [7]
- ✓ are portraits of people or animals
- ☐ only include the subject's face without the background
- ✓ are two tone (black and white) images
- ☐ use the size of stipple dots and the thickness of hatch lines to depict tone
- ✓ use stipple dots to shade the subject's face [8]
- ☐ use hatch lines to shade the subject's clothing
- ✓ place stipples so they align with facial features [38]
- ☐ do not place dots in the highlights of a subject's face
- ☐ use denser stippling for key features such as a subject's eyes, nose, and lips
- ☐ use strokes for hair
- ☐ outline the subject with strokes

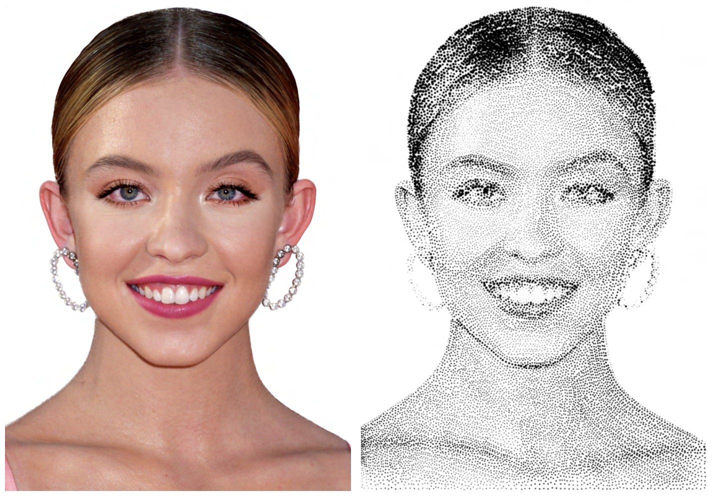


Figure 4.1: A photograph of Sydney Sweeny and the initial stipple rendering.

4.2.1 Quick Improvements on Initial Results

Although these results were exciting there were immediate issues that needed to be addressed. The first issue was that the results looked more like a comic book filter was being applied to an image, rather than it being turned into a hedcut. This was because the background of images would be included in the final drawing. Typical hedcuts, however, only include the subject. To make sure the background was not included I used Apple's "copy subject" tool [1] to extract the subject from an image and then pasted the subject onto a plain white background as shown in Figure 4.2c. The final rendering in Figure 4.2d more resembles a hedcut than the rendering in Figure 4.2b since only the subject is included. Now we can check off one more box on our list from Section 0.1.1!

Our renderings ...

[☑ only include the subject's face without the background

The next improvement that needed to be made was adjusting the size of the final dots that were placed. Stipples that were too small resulted in a rendering that was hard to see, like the cat in Figure 4.3b. Stipples that were too large resulted in the flow of the lines being lost, like the cat in Figure 4.3d. This led to the realization that every step of the hedcut process, from initial edge detection to dot placement, should be able to be parameterized.

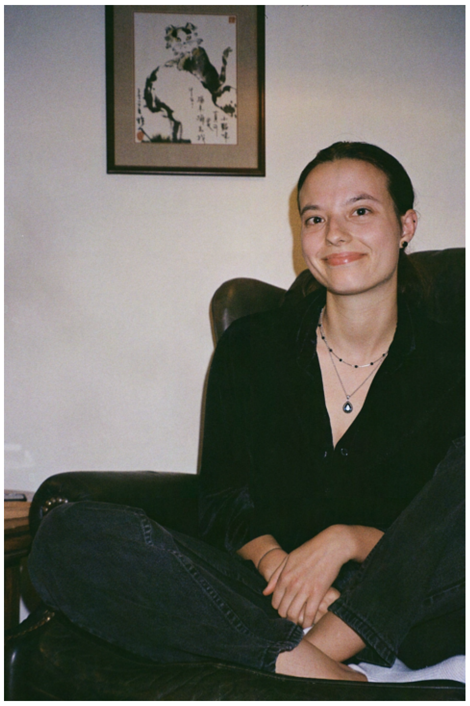
4.2.2 Creating the Command Line Interface

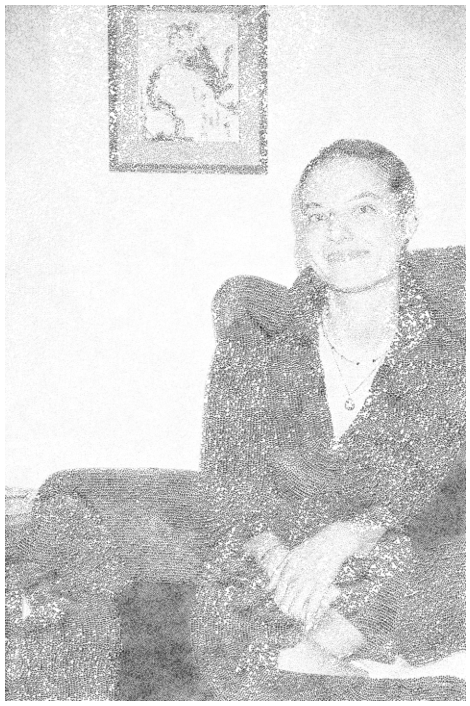
To allow the user to adjust parameters in the intermediate steps of hedcut rendering I programed a command line interface (CLI) using `OpenCV`. Tuning parameters during intermediate steps generated far better looking results than hardcoding these values. Figure 4.4 shows the improvements that can be made by adjusting the results at each intermediate step. In Figure 4.4d the dots on the forehead follow a pattern that more closely mimics hand-drawn hedcuts than the dots in Figure 4.4c.

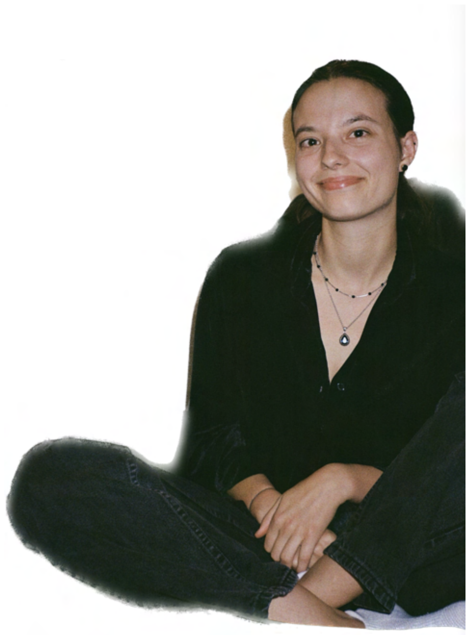
The inclusion of a CLI, or any type of user interface, is unique to my thesis. User interfaces were not present in any of the work that I surveyed. Although fully automated hedcut rendering is impressive I found that adjusting the renderings in intermediate steps (edge detection, offset map generation, initial seed placement, etc.) gave the best results. Additionally my CLI gives the user a clear understanding of the intermediate steps in this process. A video tutorial of using the CLI can be found on my GitHub.

Appendix C.3[1] gives initial renderings that include these quick improvements. With these results we can examine how well our algorithm performs on a range of input images. In renderings of subjects with lighter skin, features such as eyes and lips are more pronounced. In subjects with darker skin it is more difficult to make out these features. This could come as a result of features such as eyes and eyebrows having a similar tone to the skin. A hedcut artist, however, would be able

[1]A online gallery of these results can be found here.


(a) Me in a chair with a background.


(b) Rendering with the background.


(c) Just me, no chair, no painting.

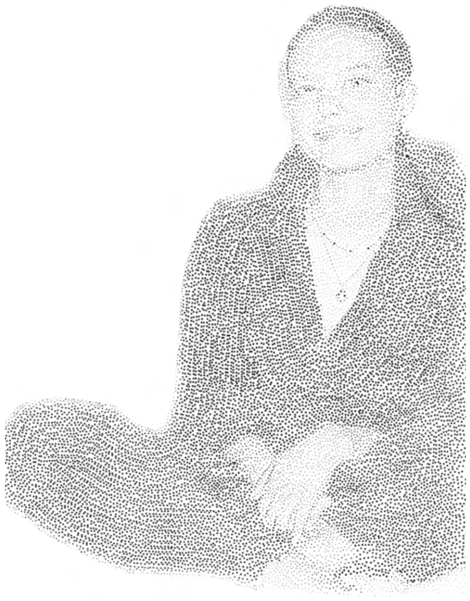
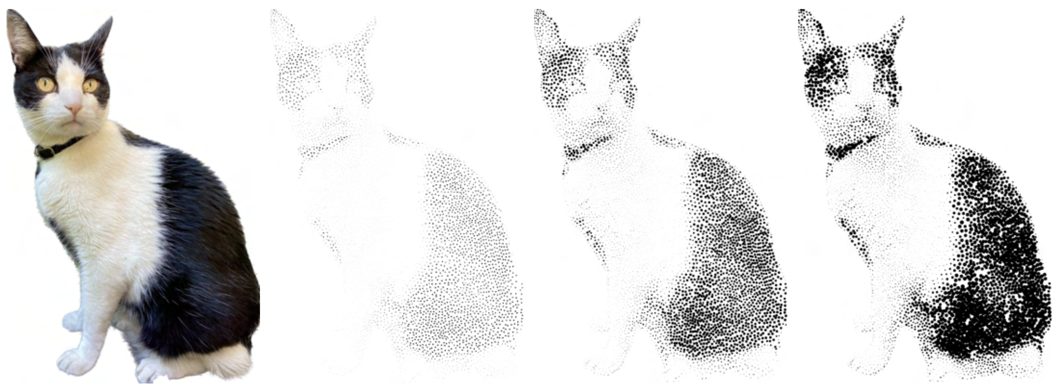
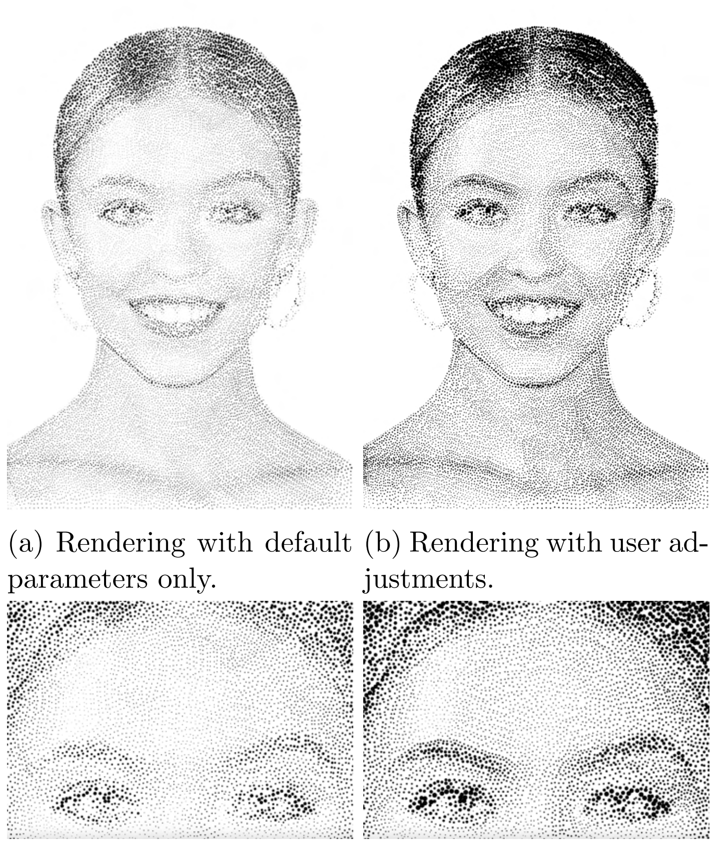

(d) Rendering of just the subject.

Figure 4.2: A rendering with and without the background.



(a) My cat, Mowgli. (b) $s_{max} = 10$ px. (c) $s_{max} = 20$ px. (d) $s_{max} = 30$ px.

Figure 4.3: Rendered stipple drawings of my cat with different maximum dot sizes.



(a) Rendering with default parameters only.

(b) Rendering with user adjustments.

(c) Close up of eyes and forehead in default paramenter image.

(d) Close up of eyes and forehead in user adjusted image.

Figure 4.4: Comparison of default parameters and user adjustments.

to properly render a darker tone by emphasizing highlights in the face and creating contrast between eyes, eyebrows, etc. and the skin around them.

4.3 Larger Improvements on Initial Results

Even with these changes I was still unsatisfied with the results. I felt that the renderings looked too real and not cartoon-ish enough. In Figure 4.5, for example, Curly has a cartoon-like quality to him. In this portrait we can see that the stipples follow long smooth feature lines along his face and jacket. In Figure 4.6, however, the feature lines are much less pronounced and there is not a clear flow in the dots. This happens due to methods used for detecting feature lines and the placement of the final dots.

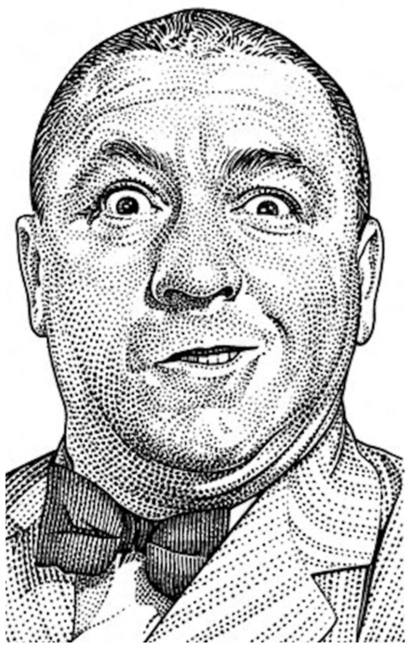


Figure 4.5: Glass, Randy. Curly Howard. [Pen and Ink]. Wall Street Journal. Retrieved from https://www.randyglassstudio.com/blog/2012/04/16/my-very-first-hedcut-for-the-wall-street-journal.

4.3.1 Improving Line Drawings

In Figure 4.7 we can see that the feature lines detected by my algorithms are rather rough. This causes the flow of features to be lost in the final rendering. If the detected feature lines do not have a smooth flow then the offset map that is generated will not include neat offset lanes. Without neat offset lanes the adjustment of the initial seed pixels will not maintain feature flow. If the adjusted seeds do not follow feature flow then the rendered dots will not align with features. Therefore to enhance feature flow in my renderings I needed to extract feature lines such that the lines are smooth and “cartoonish” rather than photorealistic and jagged.

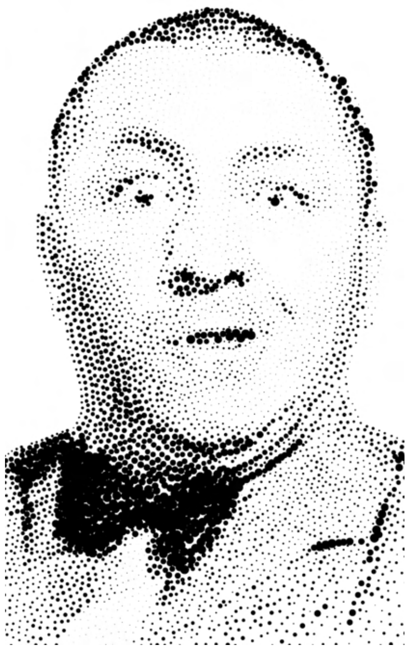


Figure 4.6: Curly Howard rendered with my program.

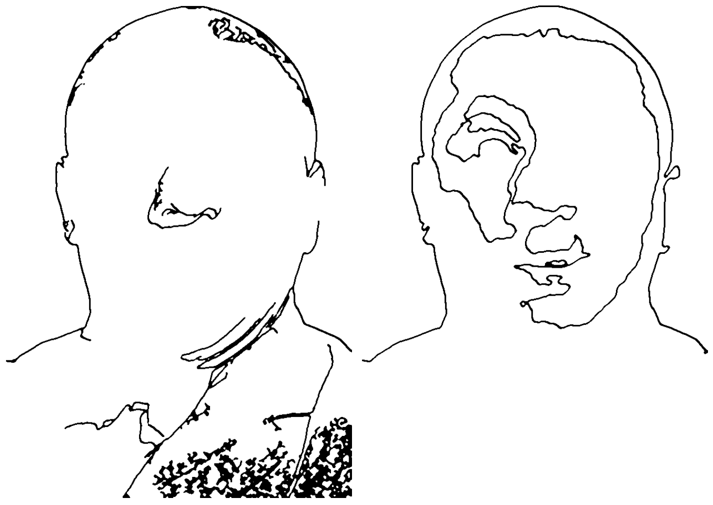


Figure 4.7: The edges and isophotesused to generate Figure 4.6

The importance of detecting smooth edges and isophotes is shown in Figure 4.8. In this example I have drawn my own edges and isophotes which are more cartoonish than those in Figure 4.7. In the final rendering we can see that the flow of the stipples is better defined and more closely mirrors that of a hand-drawn hedcut.

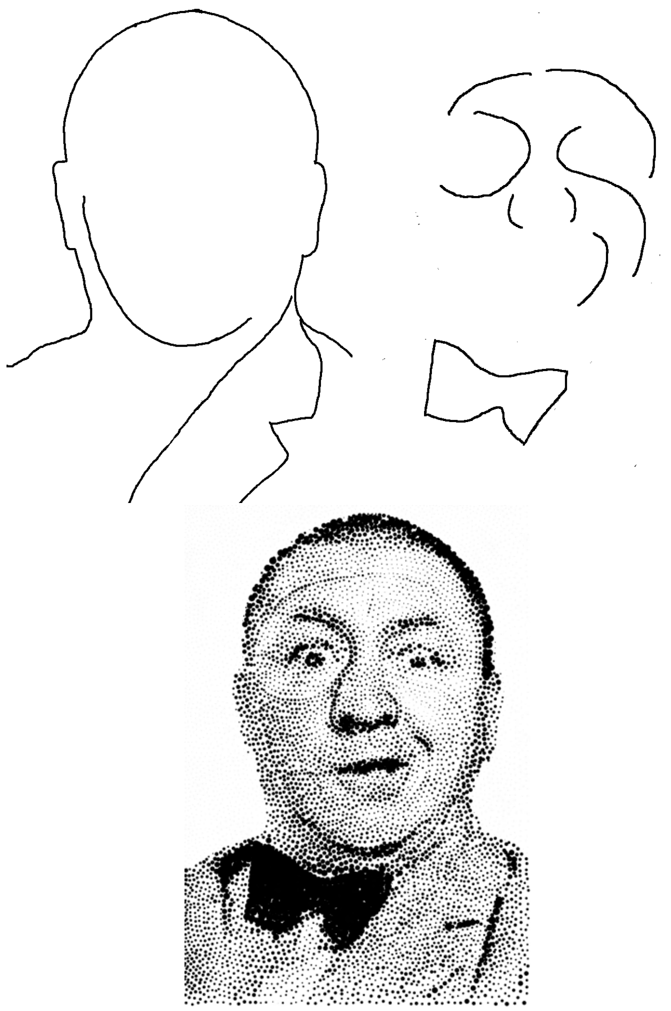


Figure 4.8: Hand-drawn edges and isophotes and the rendering they produce.

Generating smooth line drawings of faces is a field of NPR in and of itself. Popular edge detection operations such as Canny edge detection, Section 1.4.3, or the Sobel operator, Section 1.4.2, work well for detecting edges in a photograph however the results look rather photorealistic, as seen in Figure 4.9. An edge detection algorithm that results in smooth and "cartoonish" feature lines will resolve these issues and produce hedcuts that mirror those made by artists. In hand-drawn hedcuts it is clear that the contour lines that the stipple dots follow are long, smooth arcs rather than the shorter and more jagged lines often returned by edge detection algorithms.

The flow based line drawing method of Kang et al. [21] works well to produce line drawings with long, sweeping curves. Kang et al. [21] develop a novel flow-driven anisotropic filter to produce more coherent line drawings. Their work begins

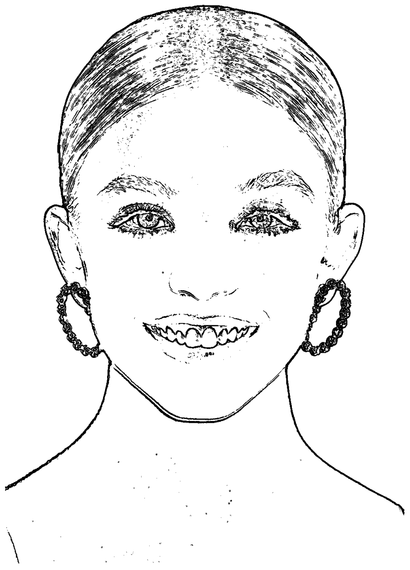


Figure 4.9: Edges detected using Sobel operator.

by applying a Sobel operator to extract an initial gradient of vectors. An edge tangent filter is then iteratively applied to expand this gradient and extract a vector field that follows the contours of the face. A flow-based Difference of Gaussians filter is applied to extract the final line drawing.

This method is used by Son et al. [36] and results in hedcuts where feature lines are longer and not jagged and where stipples align with feature lines both in the parallel and orthogonal directions. After identifying jagged and short feature lines as a problem in my renderings I attempted to implement Kang et al.'s [21] method for edge detection. However, I was unable to implement this algorithm in the allotted time. Instead I opted to make other smaller changes to the dot placement algorithms.

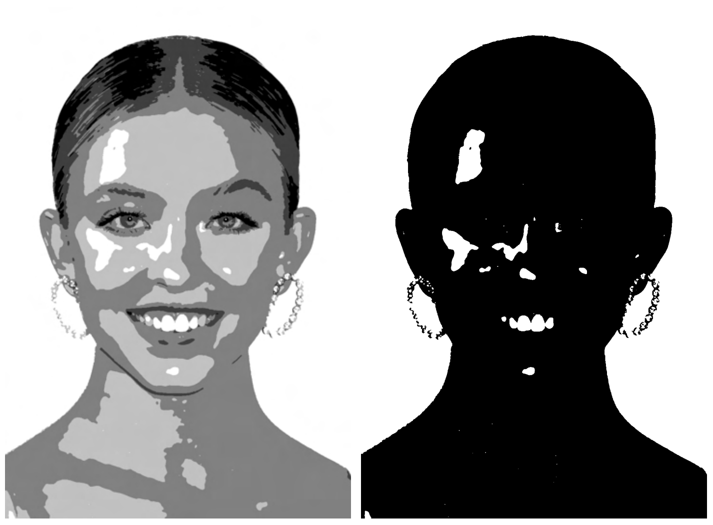
4.3.2 Modifying Dot Placement

When comparing my program's hedcuts and hand-drawn hedcuts I noticed two major differences in the placement of dots. First off, hand-drawn hedcuts typically exclude dots from the highlights of the face, such as the forehead, cheeks, and nose. Secondly, artists typically make dots denser around detailed features such as the eyes and less dense in areas line the cheeks, neck, and forehead.

Excluding Dots in Highlight Regions

Excluding dots in highlight regions required an additional step in the hedcut rendering process. Before completing step 6 in Section 0.5 we must determine which areas of the face we want to avoid stippling. Note that these sections of the face, forehead, cheeks, tip of the nose, etc. are very similar to the areas that were used to find isophotes in Section 2.2. We can follow the exact same process that we do in

Section 2.2 to generate a quantized or posterized image, Figure 4.10a. Then we can use the process from Section 2.2 to select only the lightest regions or 'highlights", as seen in Figure 4.10b. These are the regions where we will avoid placing stipples in our final renderings.



(a) Quantized regions of the face. (b) The negative space where dots won't be placed.

Figure 4.10: Selection of regions to avoid stippling.

When placing our final dots we will modify the steps in Section 3.4. For every stippled dot, *d*, we will check if *d* lies within a highlight region. If it does we will remove it from our rendering. Otherwise we will render it as we normally would. After making these changes we are able to get results like those seen in Figure 4.11.

We also improved on results by allowing for an outline to be placed around the final rendering. This was done by thresholding the initial edges such that only the longest lines, those that make up the outline of the subject, were left. We then combined these lines with the final rendered stipples. Appendix C.4[2] shows the renderings of our level 1 and level 2 input images with this technique. Now our results...

[☑ do not place stipple dots in the highlights of a subject's face

[☑ outline the subject with strokes

This modification improved renderings of subjects with dark skin because it creates contrast between eyes, eyebrows, lips, etc. and the skin around these features.

Modifying Dot Distribution

To make the distribution of dots denser around detailed features we need to modify the way in which seed pixels are placed. Recall that in Section 3.3.1 we randomly

[2]A online gallery of these results can be found here.

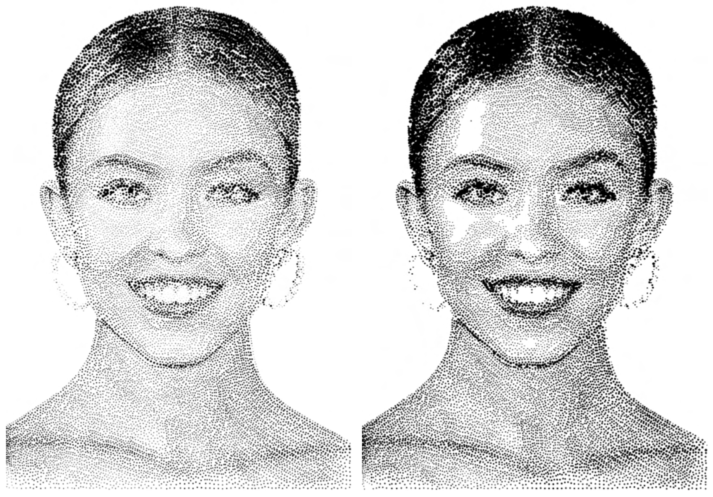

(a) Rendering with stipples placed everywhere. (b) Stipple dots excluded from highlight regions.

Figure 4.11: Comparison of rendering with and without negative space.

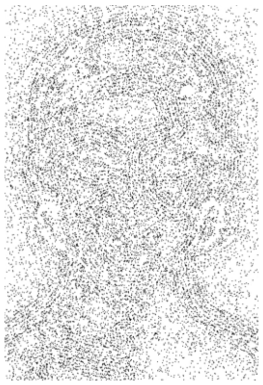
scatter dots throughout the image by scanning over all pixels, p, in the image, generating a random number r, and placing a seed at p if $r \leq \frac{1}{d^2}$ where d is the distance between offset lines. Changing this condition will change when and where dots get placed.

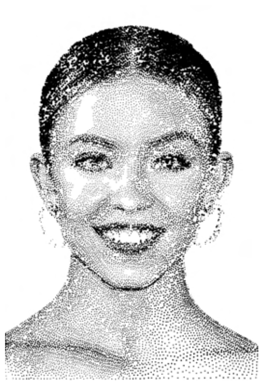
In the initial approach we will implement this modification by passing the distance map of our input image to the `placeSeeds` function, rather than just the offset map. We will create a new variable `s` which is calculated based on a pixel's distance from an offset line. As before we will place a seed pixel at p if $r \leq \frac{1}{s^2}$ where r is the same random number as before. I experimented with a variety of ways of calculating `s` and got mixed results. Figure 4.12 shows the initial seed placement and final renderings of various approaches, Table 4.1 shows the conditions that generated these dots

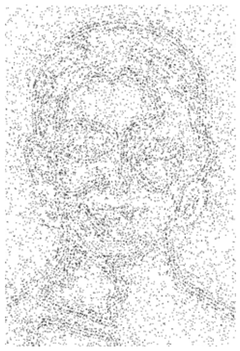
Although this method did create a greater dot density in areas closer to feature lines I did not achieve the results I was aiming for. In Figure 4.12f, for example, the subject's eyes look clear and distinct. This rendering is done with Method C from Table 4.1 which places dots very densely around feature lines. Although the eyes look decent, other feature lines, such as the isophotes around the cheeks and neck, are over-pronounced. Additionally the dots are too spaced out in the hair and the right portion of the neck.

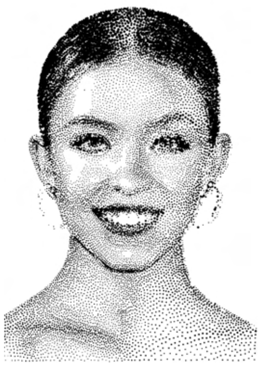
In general this approach fails because we are unable to distinguish the feature lines that make up the eyes, eyebrows, lips, etc. from the feature lines that make up the isophotes of the cheeks or neck. Distinguishing eyes from other feature lines would be very difficult without the use of AI or ML and is beyond the scope of this thesis. Our failure to write an algorithm that can distinguish different elements of the face emphasizes the skill of hedcut artists and their understanding of the complexities of human facial features.

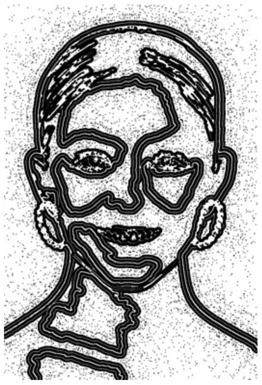
To solve this issue we will modify our CLI and allow users to select regions where

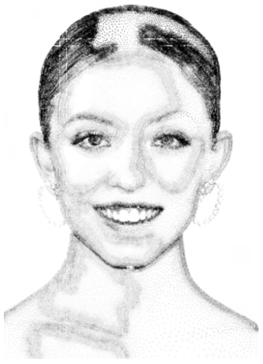

(a) Initial dots from method A†

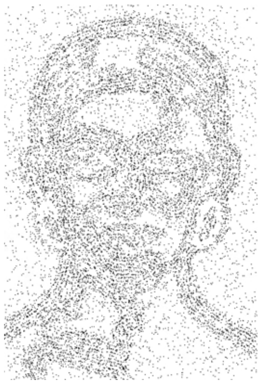

(b) Rendering of Figure 4.12a

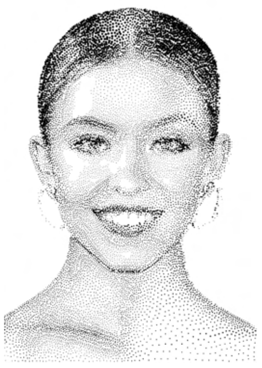

(c) Initial dots from method B†

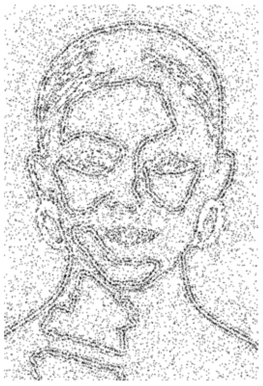

(d) Rendering of Figure 4.12c


(e) Initial dots from method C†


(f) Rendering of Figure 4.12e


(g) Initial dots from method D†


(h) Rendering of Figure 4.12g


(i) Initial dots from method E†

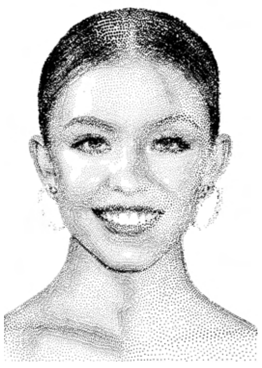

(j) Rendering of Figure 4.12i

Figure 4.12: Comparison of dot distributions.

Method	Calculation of s	Method	Calculation of s

A	d = 1
	s = d + 2 * (dist / 25)
		B	d = 1
			if dist < 10 {
			s = d * 0.75
			} else {
			s = d + 2 * (dist / 25)
			}

C	d = 1
	if dist < 10 {
	s = d * 0.25
	} else {
	s = d + 4* (dist / 25)
	}
		D	d = 2*1
			if dist < 20 {
			s = d * 0.5
			} else {
			s = d + 2 * (dist / 25)
			}

E	d = 1.5 * 1
	if dist < 5 {
	s = d * 0.5
	} else if dist < 10 {
	s = d * 0.75
	} else {
	s = d + 2 * (dist / 25)
	}		

Table 4.1: Various approaches for calculating s

they would like greater dot density using mouse actions. This was done following the `OpenCV` tutorial by Siddharth Kherada [23]. In Figure 4.13 the user has selected the eyes and the hair as regions for greater detail. We then use these regions to inform offset line spacing when generating the offset map. For this work we make offset lines in the selected regions three times as dense as the offset lines in all other regions as seen in Figure 4.14.

When we place seed pixels we give the detailed regions three times as many pixels as all other regions as seen in Figure 4.15. To account for the greater density of seeds in detailed regions we must render the final stipple dots as smaller in these areas. Suppose a pixel, p, located outside the detailed region has tone t in the input image and will be rendered as a stipple with radius r. For some pixel p' located within the detailed region that also has tone t in the input image we will give p' as radius of $\frac{r}{1.75}$. Then the rendered stipple at p' will be roughly $\frac{1}{3}$ as large the size of the stipple at p. The decision to render stipples in detailed areas as $\frac{1}{3}$ the size of stipples in non-detailed areas was made through trial and error. I made an attempt to find an exact calculation for the size of a stipple dot given its tone, t, in the input image and the density, d of dots in that region. However I was unable to complete this work in the allotted time. The final rendering with varying dot density is shown in Figure 4.16. Now we can check off one more box on our list from Section 0.1.1!

Our renderings ...

- ☑ use denser stippling for key features such as a subject's eyes, nose, and lips

If we examine the results in Appendix C.5[3] we can see that varying stipple density can help draw out details in key feature regions. In some instances, however, the reduction in detail in other areas (forehead, cheeks, neck) causes the flow of stipples and the dimension of the face to be lost. In the rendering of Obama in Figure C.7 his eyes have improved detail. It becomes difficult, however, to see a clear distinction between his chin and neck which causes his face to look flat.

Additionally the regions that should be selected for greater detail varied between subjects. In some instances selecting the lips improved the overall rendering and in other cases this caused the subject to look like they had over-lined their lips in lipstick. The issue of a certain technique working well for one subject but poorly for another came up often in this thesis. For example placing an outline around the final stipples, as seen in the renderings in Section C.5 and Section C.4, worked well when there is contrast between the subject's skin / hair and the white background. With contrast the outline was easily detectable and gave a finished look to rendering. If there is poor contrast between the subject's skin / hair and the white background (i.e. the subject has fair hair and skin) then the program does a poor job detecting this outline. These variations in results taught me that it is extremely difficult to create a single method (even with parameter tuning) that performs well for a diverse range of people.

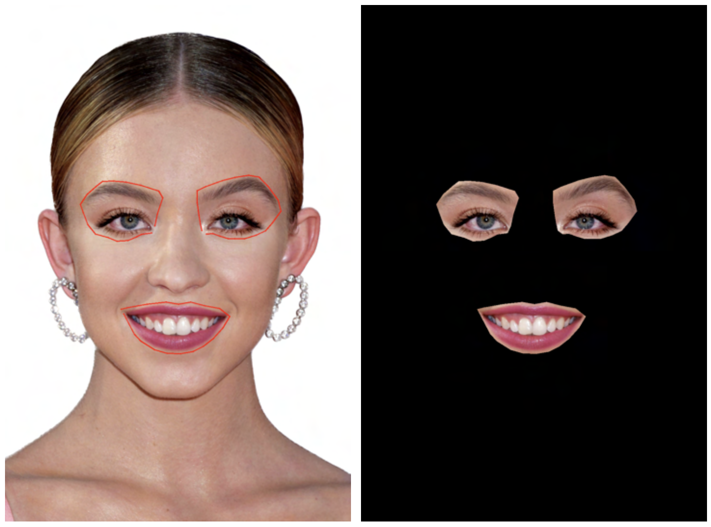


Figure 4.13: User selected details.

4.4 Suggestions for Final Approach

There are infinite ways to improve upon my program to generate results that more closely mimic hand-drawn hedcuts. We could focus on improving feature detection and line generation, finding the exact size to render stipples based on tone and density,

[3]A online gallery of these results can be found here.

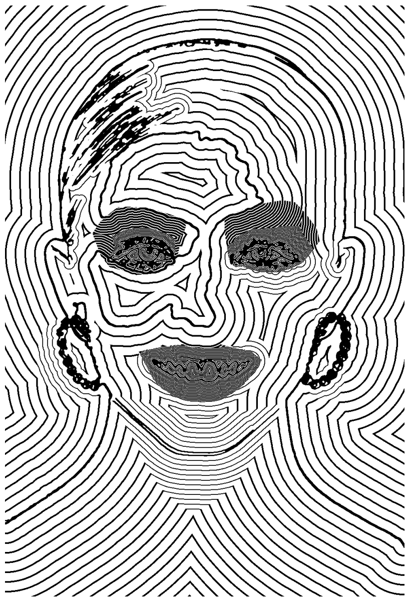


Figure 4.14: Offset lines with density variation based on user selected regions.

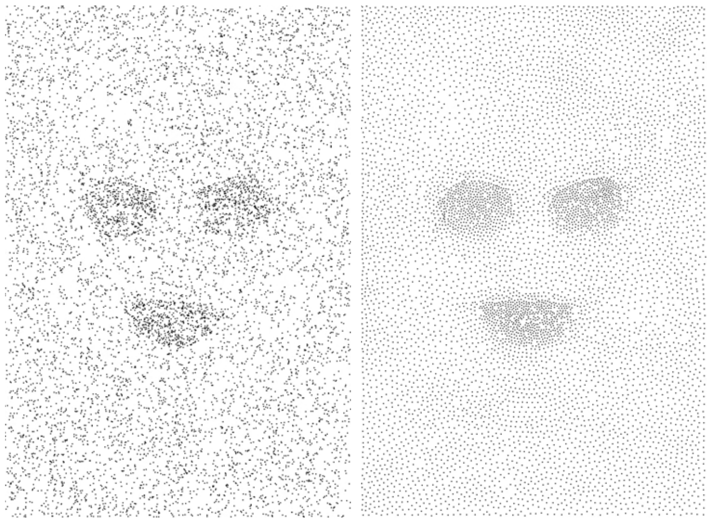


Figure 4.15: Initial and adjusted dots with detail selection.

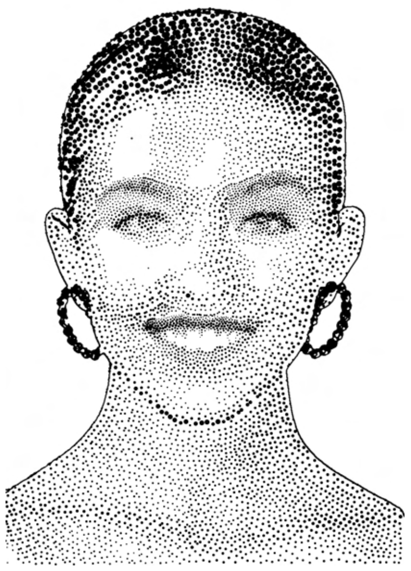


Figure 4.16: Final rendering with density variation.

rendering the dots as a more natural looking ink blot as opposed to perfect circles, using hatch lines on clothing, fading out dots in the highlight regions rather than abruptly stopping dot placement, etc. I believe the most reasonable improvements would be improving feature detection and line generation, calculating exact stipple dot size, and rendering the final dots as more natural looking ink blots. For the former this would mean using a more cartoonish edge detection algorithm, such as that described in Kang et al. [21]. Not only would this allow the dots to follow smoother contour lines but it would also improve the outline that is placed around the final rendering.

To calculate exact stipple size one could run empirical tests to compare tone generated by stipple dots of a certain density and radius and a target grayscale tone $t \in [0, 1]$. We can think of t as the percentage of white pixels that we need to achieve a given shade of gray. If $t = 0$, a pure black image, then 0% of pixels should be white to replicate this tone. If $t = 1$, pure white, then 100% of pixels should be white to replicate this tone. If $t = 0.5$, a middle gray, then 50% of pixels should be white to replicate this tone, etc. Suppose we attempted to replicate t using black stipple dots in an image with N total pixels. To see how well we replicated t we would count the number of while pixels in the stippled image, w, and check if $\frac{w}{N} \approx t$.

To render more natural looking stipples we could sample a variety of hand-drawn hedcuts and generate a dictionary with the key as dot size and the value as a list of different hand-drawn stipples of approximately size. Then for our final rendering we would calculate dots size, look up the list of hand-drawn stipples of this size, and randomly select one of them as our final dot.

4.5 Areas of Expansion for the Field of NPR

The greatest area of expansion in NPR and hedcut rendering is ensuring that algorithms perform equally well on a wide variety of input images and avoid algorithmic bias. Algorithmic bias is a "systematic deviation in algorithm output, performance, or impact, relative to some norm or standard" [14]. These biases can be moral, statistical, or social. In the case of hedcut renderings algorithms could be statistically biased if they meet the "core qualities of hedcuts" for light-skinned individuals but not for those with darker skin or if programs do poorly for faces with scarring or other abnormalities.

One may object and say that algorithms have no sense of values, that they are purely mathematical and are inherently objective. It is important to remember that algorithms implement values since they are optimized for performance relative to a standard [14]. When writing my algorithms to render hedcuts I developed the standard of "core qualities of hedcuts" and therefore had the value-laden view that certain output that aligned with these qualities was better than output that deviated from these standards. My selection of these core qualities allowed me to promote my values by implementing algorithms that performed well against these standards.

Fazelpour and Danks [14] cite problem specification, data, modeling and validation, and deployment as possible ways for introducing bias into algorithms. The first step in designing any algorithm is specifying a problem. For this thesis and other work like it our project was: "how can we use NPR to render hedcuts". These problems require "consideration of values and normative standards, and thereby [provide] a vehicle for the creation of biases" [14]. Biases are furthered if our problem specifications fail to capture real-world goals. This often happens when the group of individuals writing an algorithm are not diverse. In this thesis, for example, I was the only one writing and testing my algorithms and I frequently used input images of myself and therefore failed to capture the goal of rendering all peoples equally well.

The next area where biases could arise in NPR and hedcut rendering is data. Since our algorithms are aiming to mirror the statistics in the historical data (in this case hand-drawn hedcuts) where there are biases in the historical data there will be biases in the algorithms [4] [14]. When looking at hand-drawn hedcuts online, I found a disproportionately high number of hedcuts of light-skinned individuals as compared to those with dark skin and virtually no hedcuts of individuals with facial abnormalities. This makes it more likely that our renderings of those with dark skin or with facial abnormalities are incorrect or do not successfully mirror the methods an artist would use.

When modeling a dataset researchers will validate an algorithm's performance relative to some criteria of success [14]. As noted earlier the ways we determine this measure of "success" can promote some values over others. In the case of NPR, where a measure of success often comes down to a subjective opinion of whether or not the

[4]There have been recent efforts to detect algorithmic bias and use it as a 'signal' of previously unnoticed real world biases that should be ameliorated [14]. Perhaps the failure of the WSJ's AI to properly render bald individuals should make us aware of biases against the community without hair [8].

results look “right”, validating results can become even more biased. If researchers are more invested in producing hedcuts of those with light skin, they may be more likely to write off poor renderings of the people with darker skin as “good enough” while holding renderings of light-skinned people to a higher standard.

Finally biases can arise during deployment, in particular when the users’ values are different from those that get embedded within algorithms. If users of a NPR program to render hedcuts attempt to render images of people who do not have light skin and the algorithms produce poor results users may avoid rendering and using these images. As a result we have the potential of furthering media over-representations of people with light-skin and of continuing algorithmic biases by keeping the data set of properly rendered hedcuts small.

My work will not be used on any large scale, nor will it be used to make decisions that greatly impact individuals lives. However my work and others like it still relate to the issue of representation of a diverse range of people in the media so it is essential that we avoid these biases by creating tools that render all faces equally well. One way to avoid biases is to examine techniques artists use for making hedcuts of people with darker skin or with facial abnormalities. Son et al. [36], for example, recognized that hedcut artists use hatch lines rather than dots to depict a dark tone. Their work uses a structure grid that allows for hatching in darker regions which does well on dark skin. Bias can also be avoided by making sure programs are tested on a diverse set of images. The sets provided by Rosin et al. [34] include a wide range of skin tones which can be useful for exposing shortcomings in the renderings of certain inputs.

Appendix A

RGB to CIEL*a*b Color Conversion

OpenCV uses the following algorithm to convert from RGB to $CIEL{*}a{*}b$ color space [2]. First the RGB components are converted to floating point numbers and scaled to fit the $0-1$ range. Then X, Y, and Z are calculated

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.212671 & 0.019334 \\ 0.357580 & 0.715160 & 0.119193 \\ 0.180423 & 0.072169 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$X \leftarrow \frac{X}{X_n}, \text{ where } X_n = 0.950456$$

$$Z \leftarrow \frac{Z}{Z_n}, \text{ where } Z_n = 1.088754$$

$$L \leftarrow \begin{cases} 116 \times Y^{\frac{1}{3}} - 16 \text{ for } Y \geq 0.008856 \\ 903.3 \times Y \text{ for } Y \leq 0.008856 \end{cases}$$

$$a \leftarrow 500(f(X) - f(Y)) + delta$$
$$b \leftarrow 200(f(Y) - f(Z)) + delta$$

$$\text{where } f(t) = \begin{cases} t^{\frac{1}{3}} \text{ for } t \geq 0.008856 \\ 7.787 \cdot t \ \frac{16}{116} \text{ for } t \leq 0.008856 \end{cases}$$

$$\text{and } delta = \begin{cases} 128 \text{ for 8-bit images} \\ 0 \text{ for floating point images} \end{cases}$$

This outputs $0 \geq L \geq 100$, $-127 \geq a \geq 127$, $-127 \geq b \geq 127$. For 8-bit images the values are converted as follows $L \leftarrow L \times \frac{255}{100}$, $a \leftarrow a + 128$, $b \leftarrow b + 128$.

Appendix B

pseudocode

All code for this thesis can be found on my GitHub[1].

B.1 Thresholding Edges

```
matrix isolate (matrix m, int label) {
  for p in m {
    if p.value != label {
     m[p] = 0
    }
  }
  return m
}

bool meetsThreshold(matrix m, int threshold) {
  if (countNonZero(m) < threshold) {
    return false
  }
  return true
}

matrix skeleton(matrix m) {
  // convert m to binary

  // skeleton image and temp image
  matrix copy = m
  matrix skel
  matrix temp

  // if the image is not totally white
  if (cv::countNonZero(copy) != (copy.size)) {
```

[1]https://github.com/AriaKillebrewBruehl/senior-thesis

```
    bool done;
    do {
      open(copy, temp);
      bitwise_or(skel, !temp, skel);
      eroded.copyTo(copy);

      done = (cv::countNonZero(copy) == 0);
    } while (!done);
  } else {
    skel = copy
  }

  return skel
}

matrix threshold(matrix img, int threshold) {
  // convert m to binary

  // get components
  matrix labels
  matrix stats
  matrix centroids
  int numComps =
    cv::connectedComponentsWithStats(image, labels, stats,
    centroids);

  map<int, bool> remove;
  // for each component except the background
  for row in stats {
    int x = row[0]
    int y = row[1]
    int w = row[2]
    int h = row[3]

    // extract just the component from labeled image
    matrix comp = labels(range(y, y+h), range(x,x+w));
    // isolate component
    matrix isolated = isolate(comp, i);
    // get component skeleton
    skel = skeleton(isolated);

    remove[i] = !(meetsThreshold(skel, threshold));
  }

  for p in labels {
    if (remove[color]) {
        labels[p] = 0;
    } else {
        labels[p] = 255;
    }
  }

  return labels
}
```

B.2 Lower Envelope Algorithm

```
int f(point p) {
  int value = p.value()
  if value == 255 { // the pixel is unset
    return INT_MAX
  }
  return value
}

matrix OneDimension(matrix m, function f) {
  matrix final
  int k = 0
  array v = [0]
  array z = [INT_MIN, INT_MAX]

  // create the lower envelope
  for p in m {
    unset_p = false

    while (true) {
      // horizontal position of the kth parabola
      int r = v[k]

      if f(p) == INT_MAX {
        // this pixel is unset
        unset_p = true
        break
      }

      int s = ((f(p)+p^2)-(f(r)+r^2)) / (2i-2r)

      if s <= z[k] {
        // parabola at z[k] needs to be removed
        v.erase(v.begin() + k)
        z.erase(z.begin + k)
        k - -
      }
    }

    if unset_p {
      // pixels with unset values should not be in lower
         envelope
      continue
    }

    // update lower envelope
    k+ +
    v[k] = i
    z[k] = s
    z[k + 1] = INT_MAX
  }

  // use lower envelope to populate distance map
  k = 0
  for p in matrix {
    while z[k+1] < p {
      k + +
    }

    int a = abs(i - v[k])
    int b = f(v[k])
    int value = a^2 + b
    final[p] = value
  }
  return final
}

matrix TwoDimensions(matrix m, function f) {
  for j in m.columns {
    // extract single column from 2-D grid
    matrix column = m.column(j)
    matrix transformed = OneDimension(column, f)
    // replace column in original array with transformed
       column
    transformed.column(0).copyTo(m.column(j))
  }
  for i in m.rows {
    // extract single row from 2-D grid
    matrix row = m.row(i)
    matrix transformed = OneDimension(row, f)
    // replace row in original array with transformed row
    transformed.row(0).copyTo(m.row(i))
  }

  return m
}

matrix DistanceTransform(matrix m) {
  transformed = TwoDimensions(m, f)
  for p in transformed {
    transformed[p] = sqrt(p.value())
  }
  return transformed
}
```

Appendix C

Results

C.1 Level 1 Input Images

Rosin et al. define level 1 input images as those that ”are straightforward to stylize, [with] many restrictions [imposed]” [34]. Level 1 images are of adult faces with neutral expressions in front views only. There are clean backgrounds and no ornamentation, jewelry, or facial hair. Figure C.1 gives the level 1 input images we used for our work. Note that these images were preprocessed slightly before rendering to remove the background and crop them.

Figure C.1: All images from Pexels unless otherwise noted, reprinted and altered within Creative Commons license. From top left: cottonbro studio, Phil Nguyen, Thomas Nguka, Vodafone x Rankin, outsidethccn dsgn, Pete Souza (Wikimedia Commons, licensed under CC BY 3.0).

C.2 Level 2 Input Images

Level 2 images contain many of the restrictions from level 1, "each image contains a frontal, approximately upright, unoccluded view of a single face that fills most of the image, is cropped to include minimal clothing, and does not include hands or other body parts" [34]. Unlike level 1 images, level 2 images are allowed minimal jewelry, gaze should be mostly forward but not exclusively, and facial expressions can show more emotion. Figure C.2 gives the level 2 input images we used for our work. Like the level 1 images these images were preprocessed slightly before rendering to remove the background and crop them.

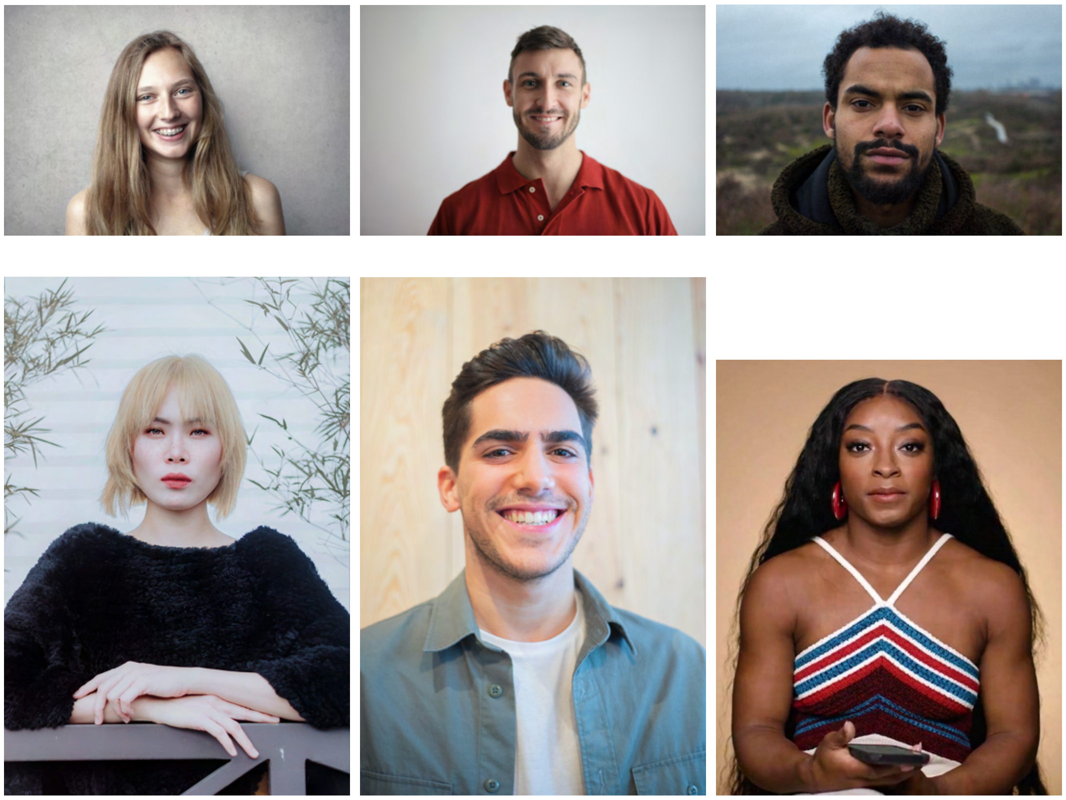


Figure C.2: All images from Pexels unless otherwise noted, reprinted and altered within Creative Commons license. From top left: Andrea Piacquadio, Andrea Piacquadio, Thyrone Paas, Eman Genatilan, Kampus Production, Condé Nast (Wikimedia Commons licensed under CC BY 3.0).

C.3 Initial Results

C.3.1 Level 1 Images

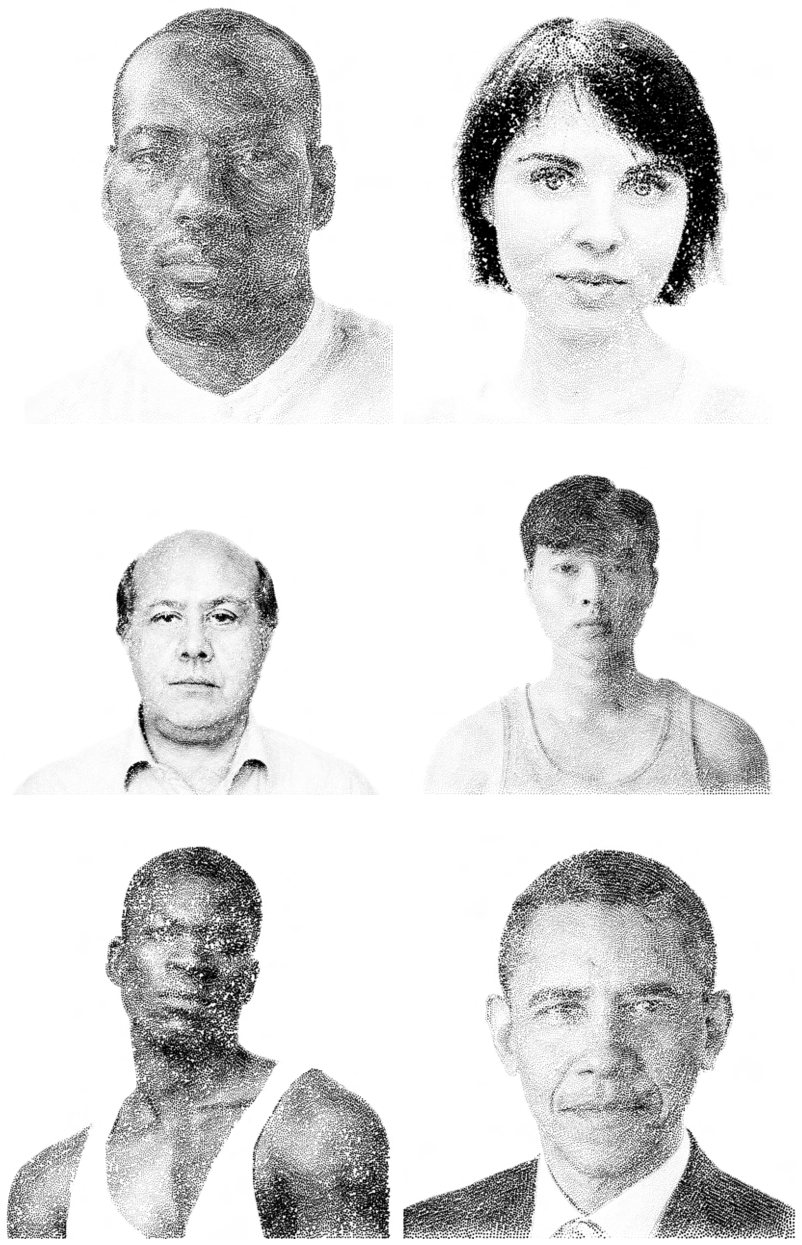


Figure C.3: Initial results on input images from Figure C.1.

C.3.2 Level 2 Images

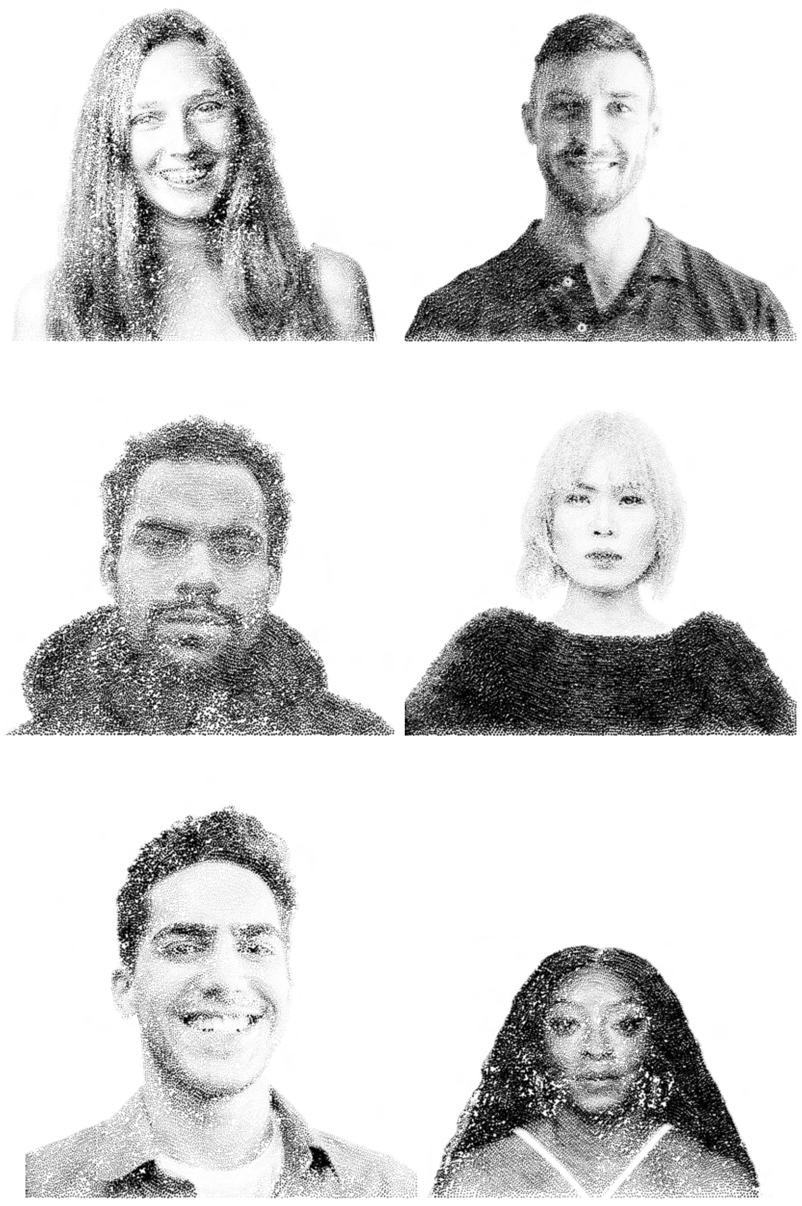


Figure C.4: Initial results on input images from Figure C.2.

C.4 Secondary Results

C.4.1 Level 1 Images

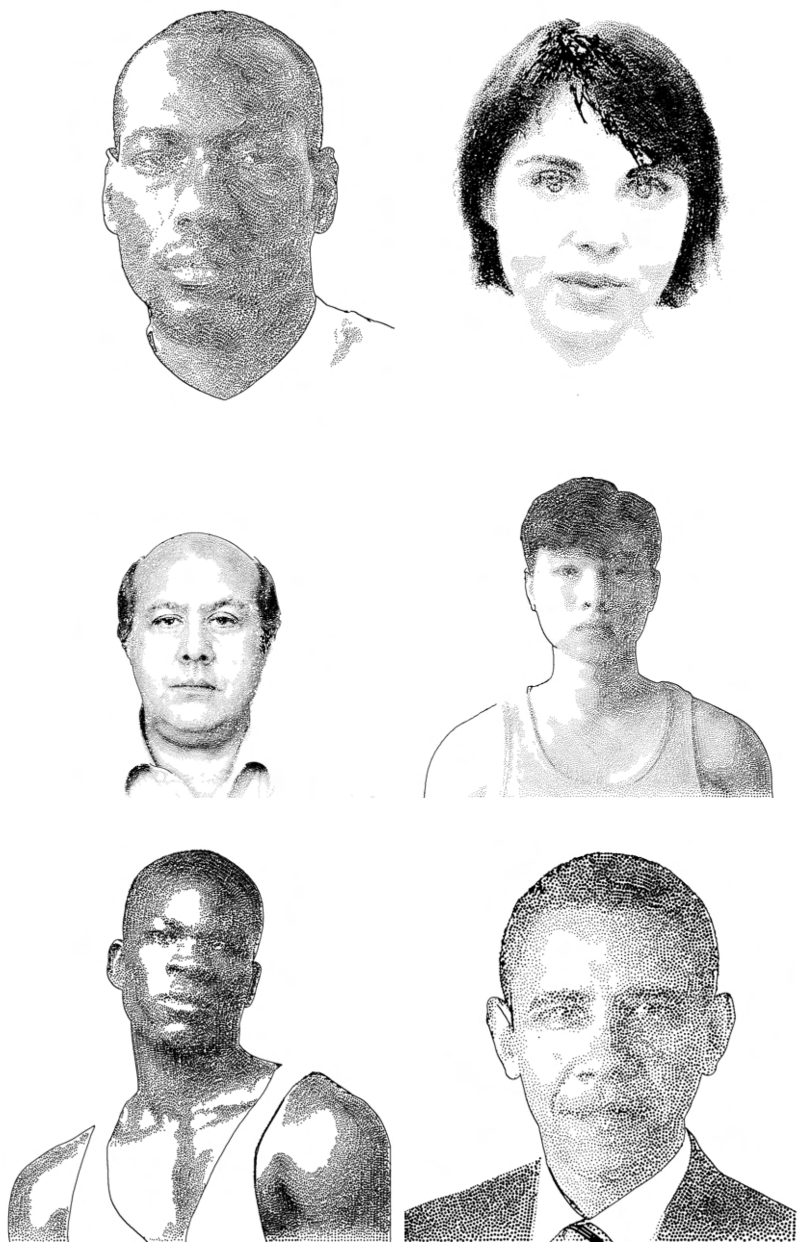


Figure C.5: Secondary results on input images from Figure C.1.

C.4.2 Level 2 Images

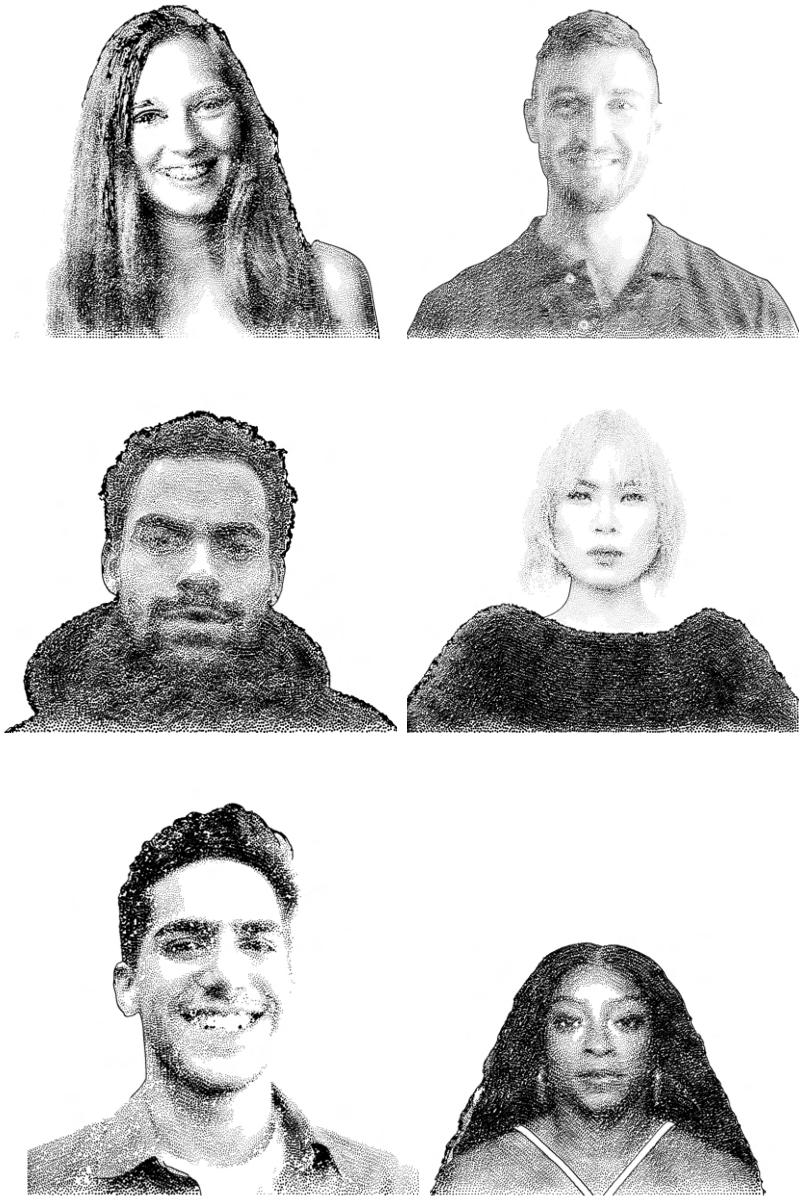


Figure C.6: Secondary results on input images from Figure C.2.

C.5 Density Variation Results

C.5.1 Level 1 Images

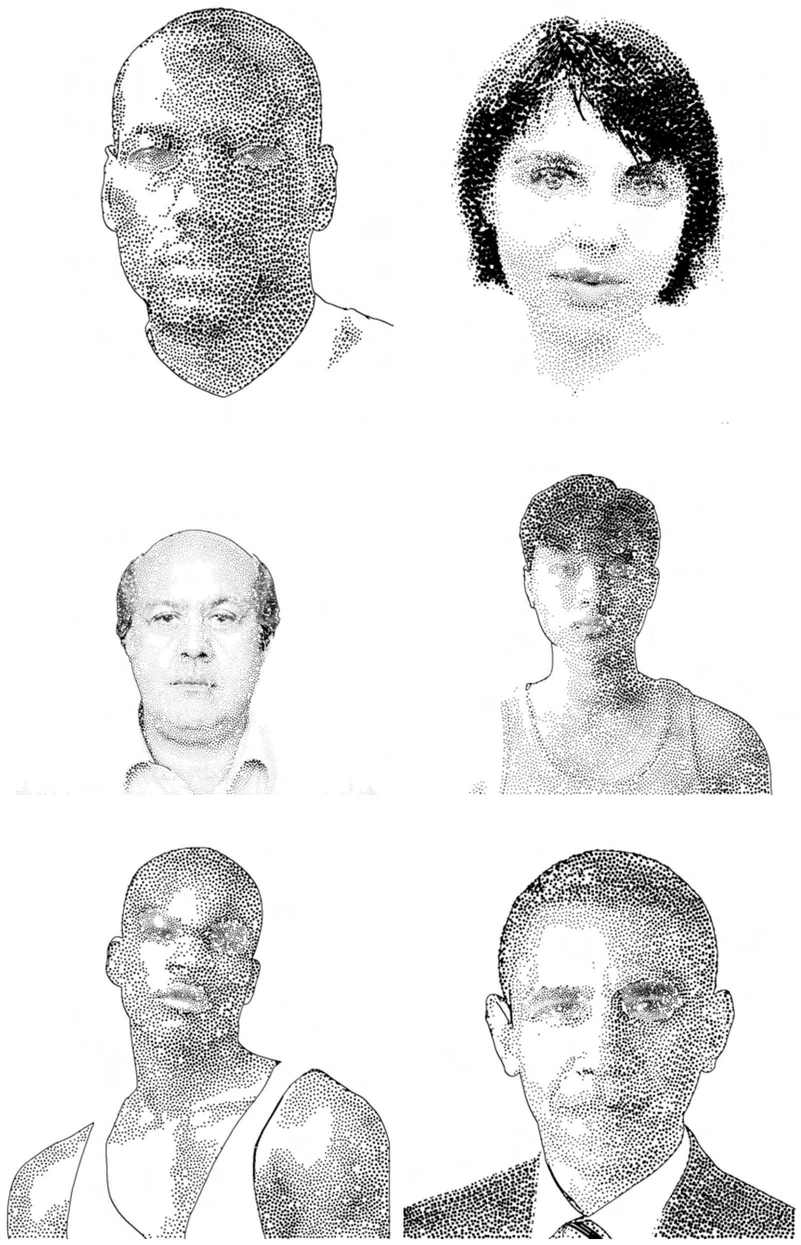


Figure C.7: Density variation results on input images from Figure C.1.

C.5.2 Level 2 Images

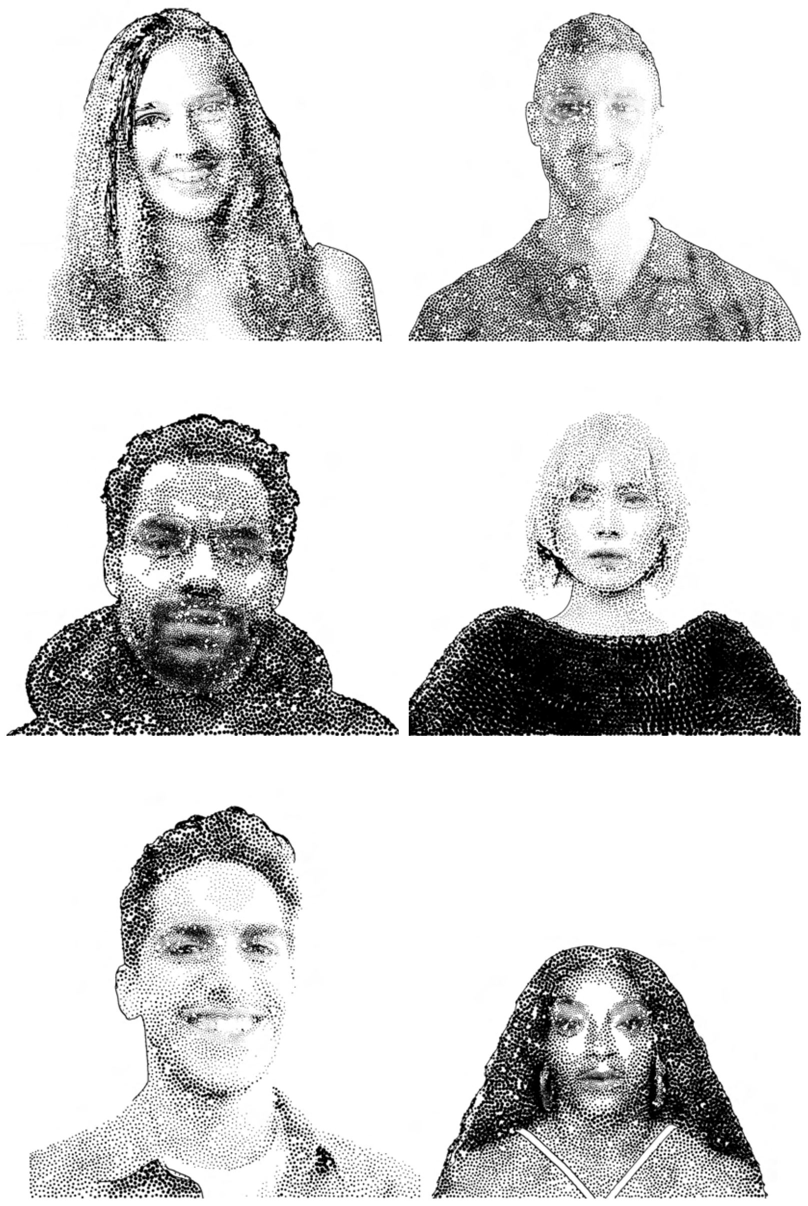


Figure C.8: Density variation results on input images from Figure C.2.

Works Cited

[1] (n.d.). Lift a subject from the photo background on iPhone. `https://support.apple.com/en-gb/guide/iphone/iphfe4809658/ios`

[2] (n.d.). OpenCV: Color conversions. `https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html`

[3] (n.d.). OpenCV: Image Thresholding. `https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html`

[4] (n.d.). What Is Image Filtering in the Spatial Domain? - MATLAB & Simulink. `https://www.mathworks.com/help/images/what-is-image-filtering-in-the-spatial-domain.html`

[5] Abecassis, F. (2011). OpenCV - Morphological Skeleton. `https://felix.abecassis.me/2011/09/opencv-morphological-skeleton/`

[6] Aguilar, B. (2010). How WSJ Stipple Drawings are Made. `https://www.wsj.com/video/how-wsj-stipple-drawings-are-made/91955BD8-9F31-4E50-AEF1-26A61B3AA2FB.html`

[7] Aguilar, B. (2019). A Short History of Wall Street Journal Hedcuts. Section: US. `https://www.wsj.com/articles/`

[8] Anderson, E., Marconi, F., & Reynolds, C. (2019). What's in a Hedcut? Depends How It's Made. Section: US. `https://www.wsj.com/articles/whats-in-a-hedcut-depends-how-its-made-11576537243`

[9] Arpin-Ricci, J. (2022). The Ethics of AI Generated Art. `https://jamiearpinricci.medium.com/the-ethics-of-ai-generated-art-57fb04b71646`

[10] Aurenhammer, F. (1991). Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Surveys*, *23*(3), 345–405. `https://dl.acm.org/doi/10.1145/116873.116880`

[11] Castleman, K. (1996). *Digital Image Processing*. Upper Saddle River, New Jersey: Prentice Hall.

[12] Chhikara, P. (2022). Understanding Morphological Image Processing and Its Operations. `https://towardsdatascience.com/understanding-morphological-image-processing-and-its-operations-7bcf1ed11756`

[13] Duke, D. J., Barnard, P. J., Halper, N., & Mellin, M. (2003). Rendering and Affect. *Computer Graphics Forum*, *22*(3), 359–368.

[14] Fazelpour, S., & Danks, D. (2021). Algorithmic bias: Senses, sources, solutions. *Philosophy Compass*, *16*(8), e12760. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/phc3.12760. `https://onlinelibrary.wiley.com/doi/abs/10.1111/phc3.12760`

[15] Felzenszwalb, P. F., & Huttenlocher, D. P. (2012). Distance Transforms of Sampled Functions. *Theory of Computing*, *8*(1), 415–428. `https://theoryofcomputing.org/articles/v008a019`

[16] Fisher, R., Perkins, S., Walker, A., & Wolfart, E. (2003). Feature Detectors - Sobel Edge Detector. `https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm`

[17] Gooch, B., & Gooch, A. (2001). *Non-Photorealistic Rendering*. A K Peters, Ltd.

[18] Halper, N., Mellin, M., Herrmann, C. S., Linneweber, V., & Strothotte, T. (2003). Towards an Understanding of the Psychology of Non-Photorealistic Rendering. In *Computational Visualistics, Media Informatics, and Virtual Communities*, vol. 11. Deutscher Universitätsverlag. `https://doi.org/10.1007/978-3-322-81318-3_9`

[19] Hughes, J., Van Dam, A., McGuire, M., Sklar, D., Foley, J., Feiner, S., & Akeley, K. (2014). *Computer Graphics Principle and Practice*. Pearson Education, third ed.

[20] Isenberg, T. (2013). Evaluating and Validating Non-photorealistic and Illustrative Rendering. In P. Rosin, & J. Collomosse (Eds.), *Image and Video-Based Artistic Stylisation*. London: Springer, 1st ed.

[21] Kang, H., Lee, S., & Chui, C. . (2007). Coherent Line Drawing. In *Proceedings of the 5th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '07, (pp. 43–50). Association for Computing Machinery. `https://doi.org/10.1145/1274871.1274878`

[22] Kasao, A., & Miyata, K. (2006). Algorithmic Painter: a NPR method to generate various styles of painting. *The Visual Computer*, *22*(1), 14–27. `http://link.springer.com/10.1007/s00371-005-0353-8`

[23] Kherada, S. (n.d.). OpenCV: samples/cpp/create_mask.cpp. `https://docs.opencv.org/4.x/db/d75/samples_2cpp_2create_mask_8cpp-example.html#a25`

[24] Kim, D., Son, M., Lee, Y., Kang, H., & Lee, S. (2008). Feature-guided Image Stippling. *Eurographics Symposium on Rendering*, *27*(4), 9.

[25] Kim, S., Woo, I., Maciejewski, R., & Ebert, D. S. (2010). Automated Hedcut Illustration Using Isophotes. In *International Symposium on Smart Graphics*, vol. 6133, (p. 12). Berlin: Spinger. `https://doi.org/10.1007/978-3-642-13544-6_17`

[26] Maciejewski, R., Isenberg, T., Andrews, W. M., Ebert, D. S., Sousa, M. C., & Chen, W. (2008). Measuring Stipple Aesthetics in Hand-Drawn and Computer-Generated Images. *IEEE Computer Graphics and Applications*, *28*(2), 62–74. `http://ieeexplore.ieee.org/document/4459866/`

[27] Mallick, S. (2015). Why does OpenCV use BGR color format ? | LearnOpenCV #. `https://learnopencv.com/why-does-opencv-use-bgr-color-format/`

[28] Niblack, W. (1986). *An Introduction to Digital Image Processing*. Denmark: Prentice Hall International (UK) Ltd.

[29] Ostromoukhov, V. (1999). Digital facial engraving. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99*, (pp. 417–424). Not Known: ACM Press. `http://portal.acm.org/citation.cfm?doid=311535.311604`

[30] Phillips, K. (2022). Additive vs. Substractive Color Models. `https://www.hunterlab.com/blog/additive-vs-subtractive-color-models/`

[31] Rhee, C.-H., & Lee, C. H. (2013). Cartoon-like Avatar Generation Using Facial Component Matching. *International Journal of Multimedia and Ubiquitous Engineering*, *8*(4).

[32] Roose, K. (2022). An A.I.-Generated Picture Won an Art Prize. Artists Aren't Happy. *The New York Times*. `https://www.nytimes.com/2022/09/02/technology/ai-artificial-intelligence-artists.html`

[33] Rosin, P. L., & Lai, Y.-K. (2015). Non-Photorealistic Rendering of Portraits. Istanbul, Turkey.

[34] Rosin, P. L., Mould, D., Berger, I., Collomosse, J., Lai, Y.-K., Li, C., Li, H., Shamir, A., Wand, M., Wang, T., & Winnemöller, H. (2017). Benchmarking non-photorealistic rendering of portraits. In *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*, (pp. 1–12). Los Angeles California: ACM. `https://dl.acm.org/doi/10.1145/3092919.3092921`

[35] Schumann, J., Strothotte, T., Laser, S., & Raab, A. (1996). Assessing the Effect of Non-Photorealistic Rendered Images in CAD. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery. `https://doi.org/10.1145/238386.238398`

[36] Son, M., Lee, Y., Kang, H., & Lee, S. (2011). Structure grid for directional stippling. *Graphical Models*, *73*(3), 74–87. `https://linkinghub.elsevier.com/retrieve/pii/S1524070310000433`

[37] Tomasi, C., & Manduchi, R. (1998). Bilateral filtering for gray and color images. In *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, (pp. 839–846). Bombay, India: Narosa Publishing House. `http://ieeexplore.ieee.org/document/710815/`

[38] Winnemöller, H. (2013). NPR in the Wild. In P. Rosin, & J. Collomosse (Eds.), *Image and Video-Based Artistic Stylisation*. London: Springer, 1st ed. `https://doi.org/10.1007/978-1-4471-4519-6`