ARTISTIC RENDERING OF PORTRAIT PHOTOGRAPHS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Eric Chih-Cheng Wong

January 1999

Abstract

In this thesis, we present a semi-automatic process to artistically render portrait photographs in a charcoal style. The software system requires the user to coach the process by identifying regions and edges in the portrait. With this information and a set of selected parameters, the user can quickly generate an artistic charcoal sketch. In general, our system produces a charcoal image by rendering the following five features of a portrait photograph: 1) the background area, 2) the hair, 3) the edges and lines, 4) the facial features, and 5) the facial tone. Although we compute the vertex points that compose the line segments in the final image, our system relies on an external software application to render these lines in a charcoal style. In order to render the hair region in the second process step above, we developed a novel image processing operation based on the Hough transform to find the orientation of each point in the hair region.

Biographical Sketch

Born and raised in the San Francisco Bay Area, Eric Chih-Cheng Wong came to Cornell University in the fall of 1992 to study computer science. As an undergraduate research assistant, Eric worked on a volume rendering and visualization project in the Program of Computer Graphics (PCG). Upon completion of his B.S. in computer science, Eric was inspired to join the PCG as a Masters student in the fall of 1996.

To my family, for always being there.

Acknowledgements

I would like to begin by thanking Professor Donald Greenberg for providing me with the opportunity to study in the Program of Computer Graphics at Cornell University. Thanks for always believing in me. It has been a wonderful experience.

During my time here, I ran into a lot of hurdles. The most difficult of these was finding a thesis topic and figuring out how to approach it. Special thanks go to Richard Coutts for always believing in my talents, for inspiring me to work in the field of artistic rendering, and for always being so excited to share and discuss ideas with me. Thank you Rich for helping me whenever you could.

I would also like to thank David Hart for always giving me great ideas. The idea of using the Hough transform to find orientation originated in his inventive mind. I would also like to thank Sebastian Fernandez for always being so helpful in answering my millions of questions about C/C++, math, signal processing, neural networks, general computer science, etc. The list must be a mile long. Thanks go to Liang Peng for being a resource when it came time for me to read about image segmentation techniques. And finally, I would like to thank Mahesh Ramasubramanian for also answering my often idiotic questions as well as sharing his positive energy with me.

As an undergraduate research assistant in the Program of Computer Graphics, I had a great time working on the CVP project. I would like to thank Philip Hubbard, James Durkin, and Gordon Kindlmann for providing me with this great experience and welcoming me to the PCG. I would also like to give special thanks go to Jeffrey Tseng for first introducing me to Professor Greenberg and the PCG.

Back when I used to work in the Architectural Modeling Group, I had a hard time adjusting and working in a free-thinking environment. I would like to thank Michael Malone, Moreno Piccolotto, Corey Toler, and Richard Coutts for helping me grow up out of my undergraduate-self, teaching me how to better speak up about my own ideas, and helping me learn how to better motivate myself.

When I was blue, Michael Malone, Richard Coutts, Corey Toler, Philip Hubbard, and Sebastian Fernandez were always there to lend an ear and to give me some helpful words of advice. Thanks guys. I owe you tons!

When it came time to get out of the lab, thanks go to Daniel Gelb for forcing me to play roller-hockey. Thanks also go to David Hart, Mahesh Ramasubramanian, and Richard Coutts for encouraging me to play tennis and dragging me to the gym.

I would also like to thank members of the wonderful staff, James Ferwerda, Ellen French, Linda Stephenson, Jonathan Corson-Rikert, Peggy Anderson, Hurf Sheldon, and Mitch Collinsworth, for always being helpful in providing me with direction in my research as well as in dealing with administrative items.

Outside of the Program of Computer Graphics, I would like to thank my family and friends for giving all of their faith and support during these past two years.

Special thanks go to JoAnn Kim for being super supportive and for helping me proof-read and make revisions of my thesis. I don't think I could have written this thesis without her support and help.

Finally, many thanks go to PCA for funding my research.

Table of Contents

List of Figures

Chapter 1

Introduction

Artistic rendering is a relatively new and broad field in the computer graphics community. Its applications are numerous, and many approaches to artistic rendering have been proposed. Although pen-and-ink style illustrations are the most common, work has also been done in many other media including oil and watercolor paints. The diversity of these approaches stems from the broad range of target media as well as the various source data types from which the art work can be generated. Furthermore, as with the work of traditional artists, artistic rendering methods often reflect the style of their authors. Yet even with this diversity, artistic rendering methods share the one common goal of *generating synthetic images that appear to have been created by a human hand.*

In computer graphics, artistic rendering is more commonly referred to as *non-photorealistic rendering.* This terminology originates from the computer graphics term *photorealistic rendering*, which involves the creation of synthetic images that are indistinguishable from photographs. However, the term non-photorealistic rendering negatively implies that the field involves the generation of all images that are not intended to look realistic as contrasted to those that appear artistic. Therefore, we prefer the more direct terminology of *artistic rendering*.

In this thesis, we present a semi-automatic process to artistically render portrait photographs in a charcoal style. The software system requires the user to coach the process by identifying regions and edges in the portrait. With this information and a set of selected parameters, the user can quickly generate a charcoal sketch.

Our system differs from other artistic rendering systems because it is the first system primarily focused on generating charcoal style drawings and is the first content specific artistic rendering system. Although this content sensitivity may seem to be a limiting factor, it actually allows for assumptions that ultimately produce higher quality drawings. For example, because we know that rendering hair will be a common task, we can develop an algorithm specifically for this purpose.

In general, our system produces a charcoal image by rendering the following five features of a portrait photograph: 1) the background area, 2) the hair, 3) the edges and lines, 4) the facial features, and 5) the facial tone. Although our system computes the vertex points that compose the line segments in the final image, our system relies on MetaCreations' Painter software [37] to render these lines in a charcoal style. An overview of artistic rendering process is presented in Chapter 3. Details of the significant image processing techniques that compose the process are discussed in subsequent chapters.

In order to render the hair region in the second process step above, we develop a novel image processing operation to find the orientation of each point in the hair region. More specifically, we define a method for computing local orientation using a line detection operator followed by multiple applications of a small Hough transform. Details of this process can be found in Chapter 4.

Unfortunately, the system we present for artistically rendering portrait photographs has limitations. Because of the assumptions that we make in the operations that compose the rendering process, our system can only handle a limited range of facial types. Primarily, our system has difficulty handling the following cases: 1) curly hair, 2) darker skin complexions, 3) dark areas on the face such as those caused by deep wrinkles or birthmarks, and 4) occluding objects to the face such as glasses.

However, within the range of facial types that are suitable for our artistic rendering process, we are able to generate convincing sketches that accurately capture the essence and the subtleties of the human facial expression. For an artist, capturing these subtleties is the most difficult part of drawing the human face. Even when directly tracing a portrait to create a sketch, it can be difficult to precisely recreate one's expression. The reason for this difficulty stems from the fact that humans are very tuned to interpreting the fine details in facial expressions. Accurately capturing these details is a significant result of our artistic rendering process. A set of sample sketches from our system is shown in Chapter 7.

1.1 Applications

As previously mentioned, the applications of artistic rendering are numerous and growing. One application for artistic rendering with high potential is in digital halftoning[1]. Grey-scale images that are rendered as pen-and-ink illustrations are often much better suited for print than if processed with traditional halftoning techniques. Photocopying pen-and-ink line drawings generally produces a much cleaner copy than a photocopy of a halftoned image.

Artistic rendering can also be used for the automatic generation of technical illustrations from 3D models. It is often much easier to extract meaning from a technical illustration than from a photograph. Lansdown *et al.* [30] give an example of car maintenance manuals to illustrate this fact. “How much use is a photograph to mechanics when they already have the real thing in front of them? Photographs often perform poorly when clear and precise delineation, explanation, and understanding is required.”

Conceptual design is yet another application of artistic rendering. Sometimes it is desirable to render a 3D model with an added level of ambiguity. For example, a designer might quickly generate a 3D sketch of a room interior, using a simple cuboid object to represent an end-table. If photorealistically rendered, the scene will be interpreted concretely, and a viewer may wonder why the designer chose to place a cube-like object in the corner of the room. In this case, the viewer misinterprets the intentions of the designer because the rendering style does not match the roughness of the design. However, if the designer could render the cuboidal object more ambiguously and sketch-like, then the lack of detail will only indicate the presence of an object rather that precisely defining what it is. The

[1]Halftoning is a process in which images are quantized into a 1-bit, black and white image. For example, halftoning is used in printing, where grey-scale images are converted to high-resolution 1-bit, black and white images.

interpretation of the object is left to the viewer. An example of artistic rendering in conceptual design can be found in Zeleznik's SKETCH system [58].

Artistic rendering is largely applicable in the entertainment industry. Cartoon style rendering is common in animated films and weekend cartoons. Furthermore, recent films, such as *What Dreams May Come* [54], use artistic rendering to create rich visual effects. Artistic rendering also has large potential in the gaming and educational software industry.

Finally, artistic rendering has also found its way into commercial applications. Viewpoint Datalab's LiveArt98 [9] is an example of clipart software that allows a user to render simple 3D geometry in an artistic fashion. Our system for artistically rendering portrait photographs is also geared toward commercial applications. Primarily, our system could be used in a digital portrait studio to generate added-value.

1.2 Thesis Organization

This thesis is organized in the following manner. Chapter 2 will begin by presenting previous and related work in the field of artistic rendering. Chapter 3 presents an overview of the artistic rendering process that we have defined. Chapters 4 through 6 then describe, in greater detail, the components that make up the overall process. Chapter 7 shows sample results and finally, Chapter 8 concludes this thesis by presenting several areas of future work in addition to closing remarks.

Chapter 2

Previous and Related Work

This chapter summarizes prior work that relates to the artistic rendering of portrait photographs. Section 2.1 first describes related work in the general field of non-photorealistic rendering. Section 2.2 then proceeds to describe work directly relating to the image processing and machine vision techniques that underlie the artistic rendering system proposed in this thesis.

2.1 Non-Photorealistic Rendering

The field of non-photorealistic rendering (NPR) is a diverse field both in the problems it proposes as well as the methods that have been introduced to solve them. Lansdown *et al.* [30] divided NPR into two general categories, *image space effects* and *perspective space effects*. Image space effects describe methods that use only an intensity-map or a photograph as source data. Perspective space effects relate to techniques that require thrree-dimensional information such as geometry or a depth map.

However, NPR is a much more complex field than that suggested by Lansdown.

More accurately, NPR can be described as a 4-dimensional matrix with axies composed of the following categories: 1) the source data types required by the system, 2) static or dynamic image generation, 3) the target artistic rendering style, and 4) the degree of user interactivity required to operate the methods. The broad range of techniques vary from user-interactive processes to generate detailed images of static photographs to automatic real-time methods for visualization.

The following section reviews the non-photorealistic work to date that is directly related to our research[1]. The majority of the discussion is structured after Lansdown. Section 2.1.1 will discuss image-based techniques while Section 2.1.2 will review techniques which assume the presence of depth or geometry information. Finally, Section 2.1.3 will review techniques of simulating natural media, an important primitive to all NPR systems.

2.1.1 Image-Based Techniques

Our proposed system for generating charcoal sketches from portrait photographs is categorized as an image-based technique. The works presented here have provided much of the motivation and the inspiration behind our research.

Haeberli [19] describes several methods for generating painterly[2] images from a source photograph (Figure 2.1a). In the most basic form, Haeberli introduces a technique that allows a user to interactively place brush strokes onto a image. The

[1]A more comprehensive listing the published research in the field of NPR can be found in the bibliography.

[2]Painterly is defined by the *Marriam Webster* dictionary to be "of, relating to, or typical of a painter." In the non-photorealistic rendering literature, painterly generally refers to techniques that simulate an impressionist style of painting.

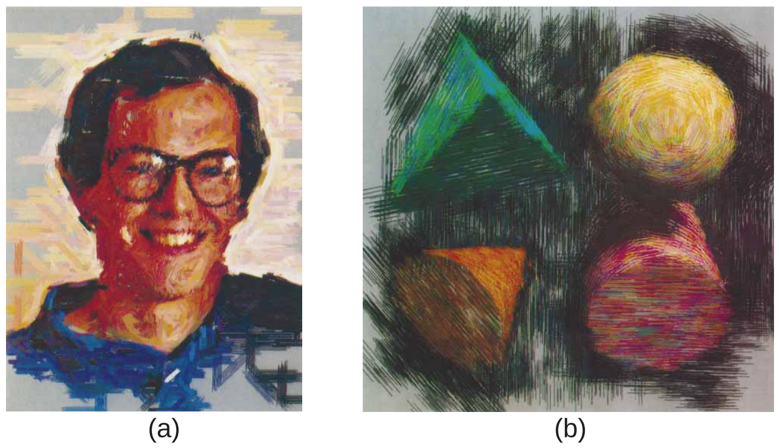


Figure 2.1: *Heaberli. [19] Two sample results of Haeberli's system. Figure (a) shows a painting with multiple sized brushes and strokes oriented along the source image gradient. Figure (b) shows a sample of Haeberli's system working in conjunction with a ray-tracer.*

colors of the strokes are sampled from the underlying source image rather than being chosen by the user. The orientation and the size of the brush strokes can be determined either by the gradient direction of the source image or by interactive techniques, such as using mouse direction and speed to determine stroke orientation and size. Haeberli's system also allows various brush stroke types to achieve both painterly and abstract results. In addition to generating painterly renderings from source photographs, Haeberli also briefly describes using his system in conjunction with a ray-tracer to abstractly render geometry (Figure 2.1b). To do this, Haeberli samples information from the geometry, such as color and surface normals, to orient the strokes in the final painting. A method for using relaxation to automatically generate images is touched on as well.

Salisbury *et al.* [43] present an interactive system for generating pen-and-ink style illustration by allowing users to interactively paint with *stroke textures.*

Salisbury defines a stroke texture as "a collection of strokes arranged in different patterns." Figure 2.2a shows some sample stroke textures. To operate the system, a user first selects a stroke texture and a target tone. Then with these selections, the user can "paint" the illustration, while the computer is left responsible for drawing the individual strokes. In addition to interactively painting the tone map, the user can also allow the system to sample tone from a reference image. A grey-scale image is imported and its pixel values are used as tone. Salisbury describes two methods for matching painted textures to tones. For more basic textures, the system can randomly apply strokes from the source texture to the target image until the correct tone is met. For more complex textures, such as cross-hatching, the user is responsible for prioritizing the individual strokes in the texture. Strokes with a higher priority are drawn before lower priority ones when rendered (Figure 2.2a). Lastly, Salisbury describes the use of *procedural stroke textures*. These textures are automatically generated and can be oriented along the gradient direction of the underlying reference image. Results of this technique are show in Figure 2.2b.

Pnueli *et al.* [40] introduce a system for both automatically and interactively creating halftoned images that resemble man-made engravings[3]. In general, their technique works by computing a set of "equipotential" lines from an image. Pnueli defines equipotential lines as curves in an image that are separated by a constant delta, defined by the user, in gradient-magnitude space. These lines produce an

[3]Halftoning is a process in which images are quantized into a 1-bit, black and white image. For example, halftoning is used in printing, where grey-scale images are converted to high-resolution black and white images.

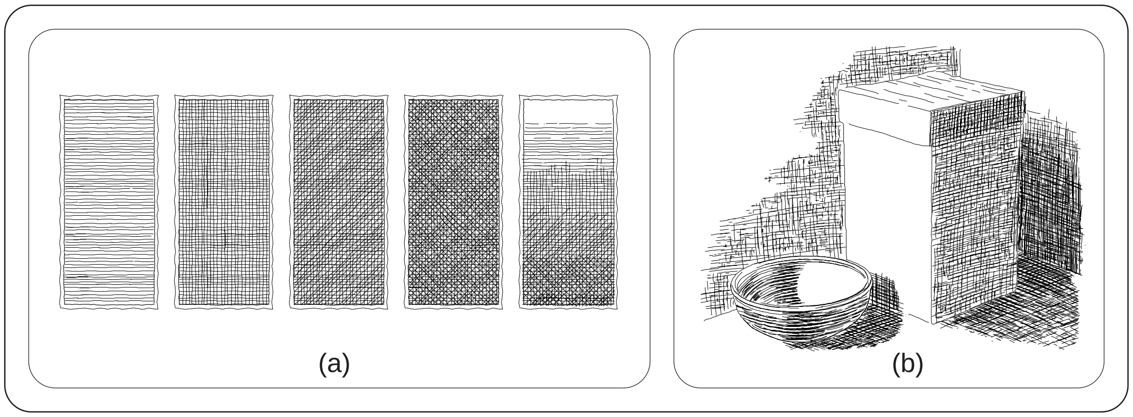


Figure 2.2: *Salisbury et al. [43] Figure (a) shows a sample prioritized stroke texture. Note how horizontal lines are drawn before vertical lines, vertical lines before diagonal lines, and so on. Figure (b) show some results of Salisbury's illustration system.*

image that resembles a man-made engraving. Although the system can automatically generate a set of these lines for any image, the user can choose to use a more interactive mode. As a first step, the user must subdivide the source image into regions. Each region is then individually halftoned with a different set of parameters chosen by the user. The halftoned images are then composited to produce the final engraving.

Sherstinsky *et al.* [47], [48], [49] introduce a new method for automatically halftoning photographs to create the appearance of the "hand-drawn" style found in the *Wall Street Journal* using the *M*-lattice system[4]. Sherstinsky states that the *M*-lattice "is a non-linear dynamical system that is well-suited for a variety of applications formulated as constrained non-linear optimization. In particular, it can perform image processing operations that emphasize oriented patterns." To

[4]Sherstinsky notes that the process of producing a hand-drawn halftoned image, such as those that appear in the *Wall Street Journal*, takes an artist roughly 3-5 hours.

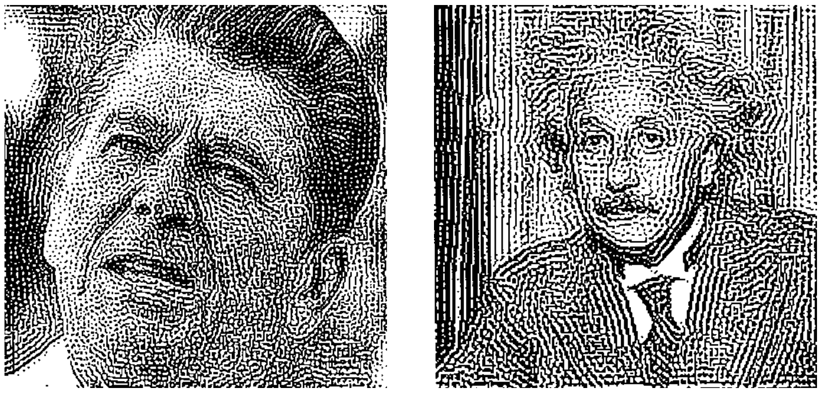


Figure 2.3: *Sherstinsky et al. [47], [48] A few results of Sherstinsky's orientation-sensitive halftoning system. Note how the dots in the halftones form lines that follow the orientation of major features.*

find this orientation information, Sherstinsky employs steerable filters [15], which are used to guide the action of the M-lattice system. Results of Sherstinsky's system are shown in Figure 2.3.

Salisbury *et al.* [45] describe "a representation for pen-and-ink illustrations that allows the creation of high-fidelity illustrations at any scale or resolution." In other words, Salisbury introduces a technique to allow the scaling of pen-and-ink illustrations without changing their apparent tone or artistic quality. A new re-sampling algorithm that relies on a low-resolution tone map and a high resolution discontinuity edge map underlies their technique. When an illustration is resized from a lower to higher resolution, a new intensity map is computed by interpolating pixels from surrounding values. However, unlike a standard re-sampling algorithm, Salisbury's technique does not weight pixels that are located across edge boundaries (Figure 2.4c-f). Once the tone map has been scaled, Salisbury picks strokes from

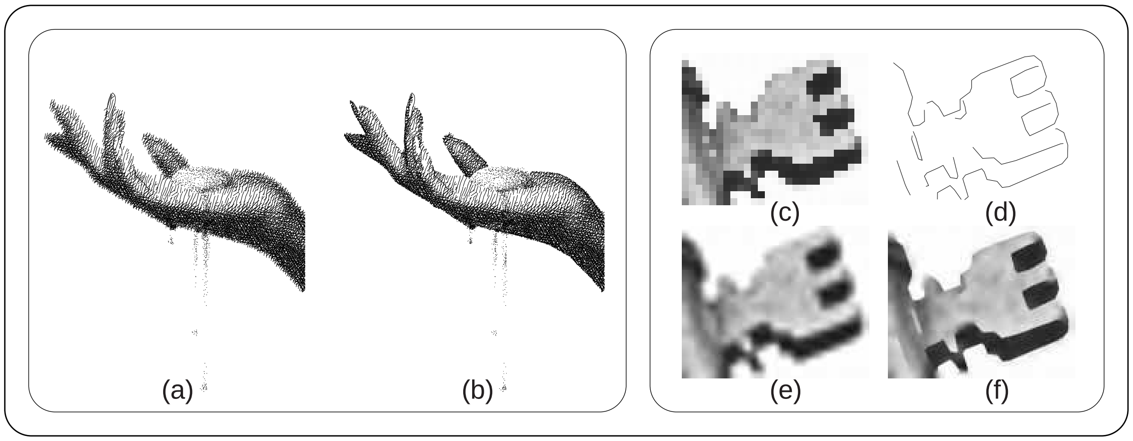


Figure 2.4: *Salisbury et al. [43] Figures (a),(b) show the results of "blasting" strokes on low-resolution tone map scaled with and without a discontinuity edge map. The results using the new representation, Figure (b), are noticeably sharper. Figures (c)-(f) illustrate the advantages of using the discontinuity edge map. Figure (c) is the original low resolution tone map. Figure (d) is the edge map. Figure (e) is the original tone map scaled via standard interpolation. Figure (f) shows the results of the new scaling technique.*

a stroke texture and applies them to the resulting image until the target tone is met. Salisbury refers to this method of placing strokes as *blasting*. Figure 2.4a,b shows a sample result.

Litwinowicz [34] describes a method for generating painterly animations from video sequences. The basis of his technique is similar to that of Haeberli except that Litwinowicz adds additional algorithms to handle the frame-to-frame coherence necessary for video. Like Haeberli, Litwinowicz achieves a painterly style by sampling colors from a source image, and then for each sample, paints a stroke onto the final image. To generate smoother and more temporally coherent animations, Litwinowicz applies optical flow techniques. These techniques enable paint strokes to track their associated objects in the source video. Because paint strokes move

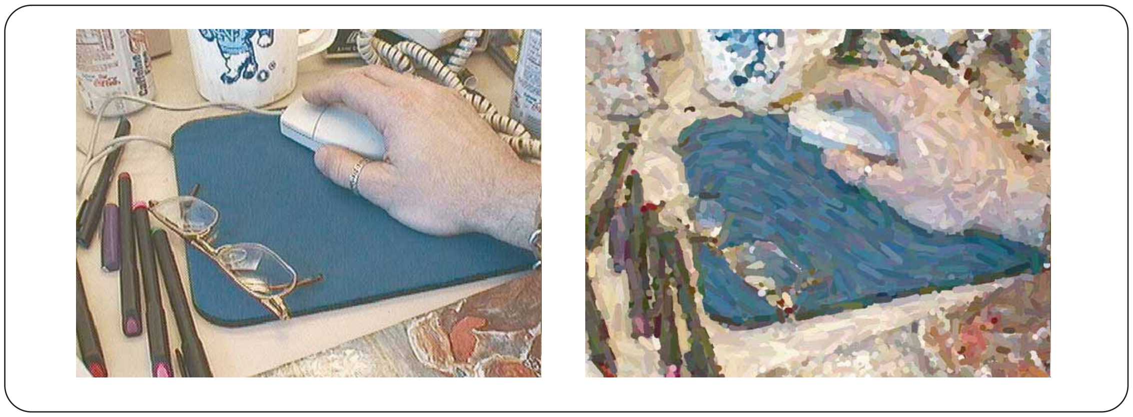


Figure 2.5: *Litwinowicz. [34] The image to the right illustrates the results of a single animation. The image to the left is the source frame. Note how the orientation of the strokes on the mouse pad are consistent with the surrounding stroke orientations.*

over time, special care is taken to introduce or remove strokes where stroke populations become too sparse or dense. Orienting strokes based on gradient information from the corresponding video frame provide additional coherence. Rather than using a standard gradient, Litwinowicz introduces a novel method for interpolated gradient values from surrounding values in regions where the gradient magnitude is low. For instance, in flat colored regions, strokes are oriented in a consistent, rather than haphazard, direction to their surroundings. Figure 2.5 shows the results of this technique.

Salisbury *et al.* [44] describes a method for interactive pen-and-ink illustration using orientable textures. In their system, Salisbury creates a vector painting program which allows a user to interactively draw vector fields over a source image. These vector fields are then used to orient small collections of B-splines or *orientation textures* which are then composited and drawn onto a final image. Figure 2.6a illustrates this process. As in their previous work [43, 45], careful attention is

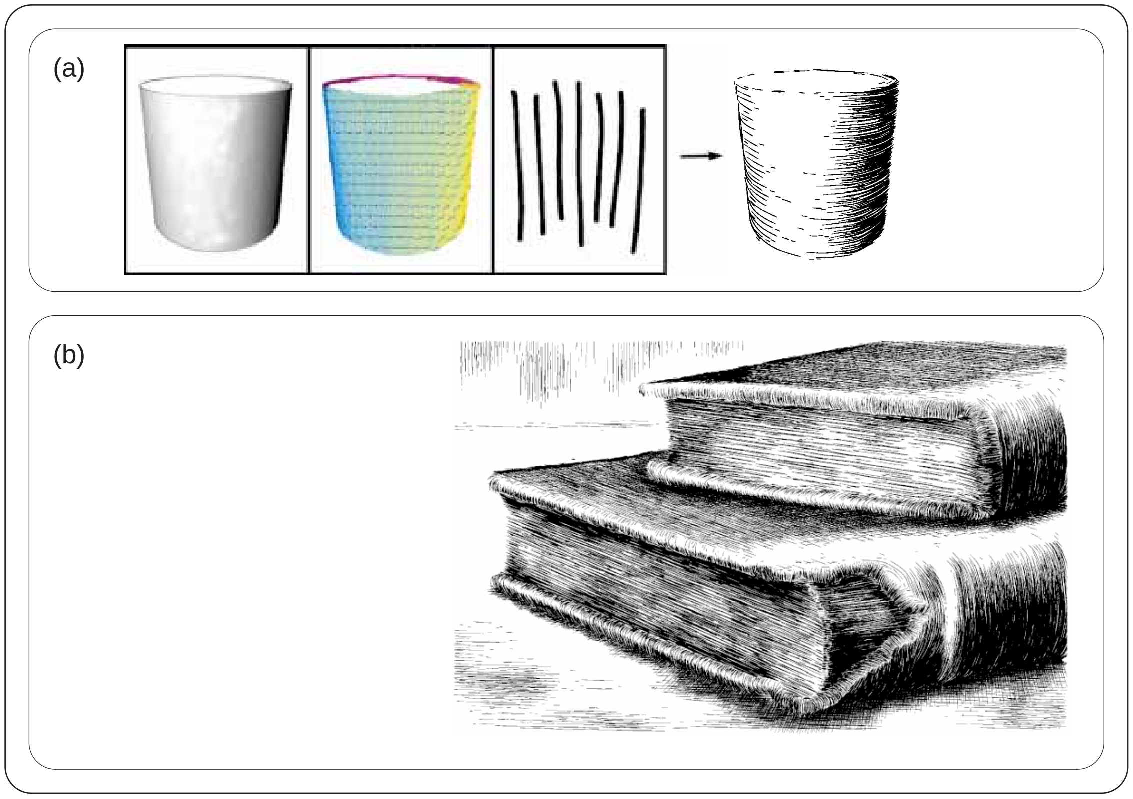


Figure 2.6: *Salisbury et al. [44] Figure (a) illustrates the process of generating a pen-and-ink illustration. The first image is the source tone map. The second image is the vector field painted by the user. The third image is the orientation texture. The final image is the resulting illustration. To construct this image, the orientation texture is rotated in the proper direction defined by the vector field. The texture is then placed into the final drawing with care taken to match the tone of the source tone map. Figure (b) shows a more detailed result.*

given to the rendering of strokes so that they do not cross object boundaries and so that they preserve the tone of the original image. Figure 2.6b shows a sample result.

Most recently, Hertzmann [20] introduces another extension to Haeberli's work by adding curved brush strokes and the automatic generation of painterly renderings. Hertzmann defined a coarse to fine technique for painting where a set of large

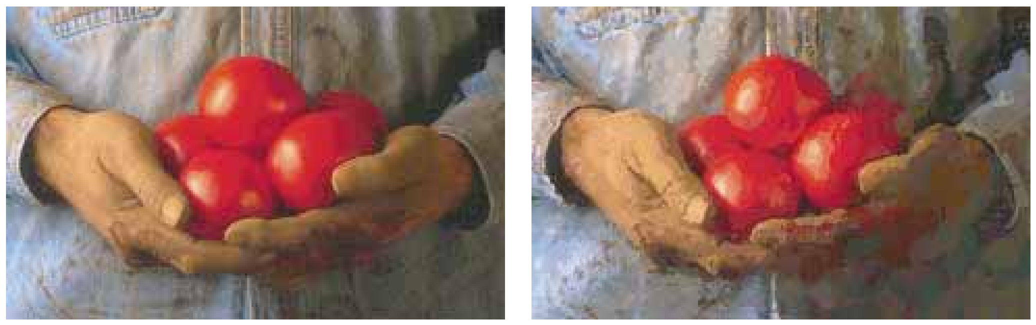


Figure 2.7: *Hertzmann. [20] A sample result of Hertzmann's system is shown on the right. On the left is the the source image.*

strokes are painted before successively smaller and smaller sets. Between each set, errors are computed between the current painting and the source image. Smaller strokes are then added for areas of large errors. This process is repeated for three to four iterations. Hertzmann also extends Haeberli's work by introducing curved brush strokes which are painted along lines perpendicular to the direction of the maximum gradient. Figure 2.7 shows a sample result.

2.1.2 Geometry-Based Techniques

Although geometry-based NPR techniques assume different underlying data types, the various approaches introduced to solve these problems often share many of the same concepts as those described in the previous section. The following section will discuss some of the major and more relevant geometry-based techniques to this thesis. This section is not an exhuastive survey, however, and the reader is invited to review other related works [11], [12], [18], [32], [14], [29].

Saito *et al.* [42] describes a method for generating pen-and-ink style illustrations from 3D geometry using image processing techniques. In general, their method generates a set of 2D geometric-buffers, termed a *G-buffer*, to which both standard and novel image processing operations are applied to create a final pen-and-ink style illustration. The G-buffer is similar to a standard Z-buffer, except that it contains additional information rather than just depth. The G-buffer includes the *nx*, *ny*, and *nz* buffers which contain scalars that represent the angle of separation between an object's surface normal and the x, y, and z axies respectively. For example, a high bin value in the *nx* buffer represents a point on the surface of an object whose surface normal is directed along the positive x-axis. A low bin value represents a point whose surface normal is directed away. The G-buffer also includes a standard Z-buffer that Saito refers to as the *sz* buffer. Finally, Saito adds the *ou* and *ov* buffers to the G-buffer that represent the u and v object patch coordinates respectively. With these buffers and a few image processing techniques, Saito describes how to generate a pen-and-ink style illustration. The general tone of the final image can be computed by blending the *nx*, *ny*, and *nz* buffers. Applying differential operators on the *sz* buffer, such as an edge-detector, produces edge information. The *ou* and *ov* buffers are used to compute hatching information via a novel image processing filter.

Winkenbach *et al.* [55] describes a method for generating convincing pen-and-ink illustrations from 3D geometric models. Their technique, in conjunction with the work done by Salisbury *et al.* [43], uses the concept of *stroke textures* as a basis for their NPR systems. In short, Winkenbach describes a system for applying

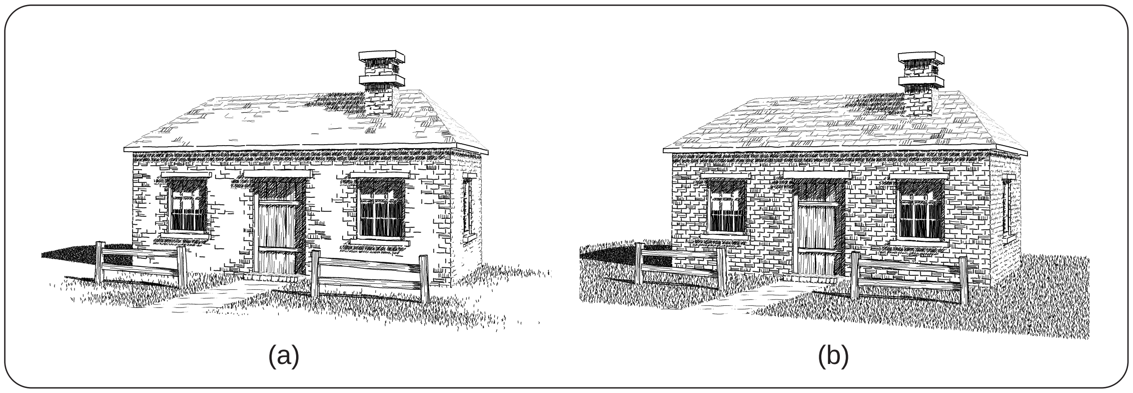


Figure 2.8: *Winkenbach et al. [55] Figure (a) shows the results of Winkenbach's illustration system. Figure (b) shows the same rendering without the use of indication.*

stroke textures as texture maps onto the geometry of a polygonal model to generate pen-and-ink style illustrations. Although simple in concept, Winkenbach discusses many technical issues necessary to produce more convincing results and faster running times. For example, *prioritized stroke textures* allow the system to render stroke textures into the final image with the appropriate tone. These textures work by requiring the strokes of a texture to be prioritized and requiring the system to draw the strokes from the prioritized stroke texture (highest priority first) until the target tone is met. Winkenbach also introduces a system for simulating *indication*. Indication is a technique used by artists in which only small portions of a texture are illustrated to imply or "indicate" that an entire surface is made of that texture (Figure 2.8). Futhermore, *boundary outline textures* are used to create object outlines, and special methods are introduced to minimize them in the final rendering. Figure 2.8 shows a sample result of Winkenbach's work.

Winkenbach *et al.* [56] later extended their earlier work [55] to allow for parametric surfaces. Their previous work had only allowed for polygonal meshes. To solve this problem of rendering parametric surfaces, Winkenbach introduces the concept of *controlled-density hashing*. This algorithm takes curves represented in the parameter space of the surface and renders them in the image domain such that the resulting tone of the projected curves form the correct tone (Figure 2.9a). Winkenbach develops a method for incrementally determining the image-space distance between two curves on the surface of an object to accomplish this. To draw the strokes, the user first sets an initial distance between a set of strokes in parameter space. The software then traverses down these strokes, computing distances between them, and adding/removing strokes where distances are either too large/small to match the target tone. Winkenbach also takes special care in computing a planar map from which strokes are clipped. The planar map contains object edge boundaries and thus, edge information that strokes should be clipped against. Finally, Winkenbach explains how a *shadow planar map* can be used to create an edge map from which shadowed regions can be created. Results are shown in Figure 2.9b.

Meier [36] introduces a method for generating animated painterly renderings from 3D geometric objects. Essentially, Meier's technique works by sticking paint strokes to the surface of objects. These strokes are then transformed and projected into screen space to generate a resulting painterly image. Animation is achieved by simply rendering successive frames of the model. Frame-to-frame coherence is maintained because the paint strokes are associated with the underlying geometry and not screen space coordinates.

Meier's technique begins by tessellating the surface geometry into triangles.

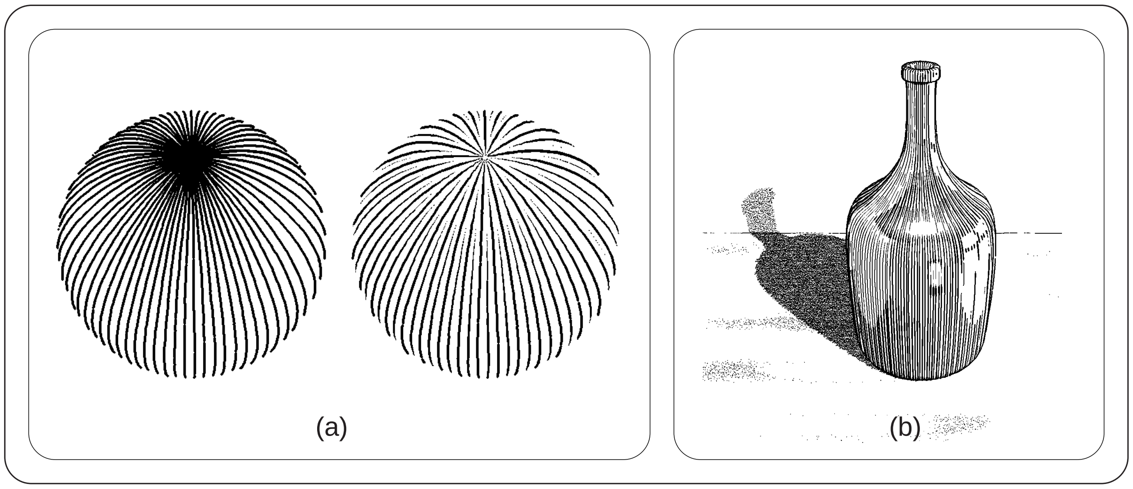


Figure 2.9: *Winkenbach et al. [56] Figure (a) shows how an evenly toned image can be generated via controlled density hatching. Figure (b) shows a final result of Winkenbach's NPR system.*

Then, for each triangle, the surface area is computed, and the triangle is populated with particles. Their number is proportional to the area of the containing triangle. The original model is then rendered using various shaders. Standard smooth-shading techniques generate a color-buffer. An orientation buffer is then computed by projecting the surface normals of the object in the direction of the view vector or any other specified vector. Finally, a scaling buffer, which holds information on how to size the paint strokes, is generated. Meier notes that because the scaling buffer is simply an intensity map, lighting, texture maps, or specialized shaders can be used to generate this buffer. Once these buffers have been generated, the particle model is transformed into screen coordinates and paint strokes are drawn from back-to-front. A stroke's color, orientation, and size are determined from the associated (x, y) values in the corresponding buffer. The overall system is summarized in Figure 2.10a, and a sample sequence of images is shown in 2.10b.

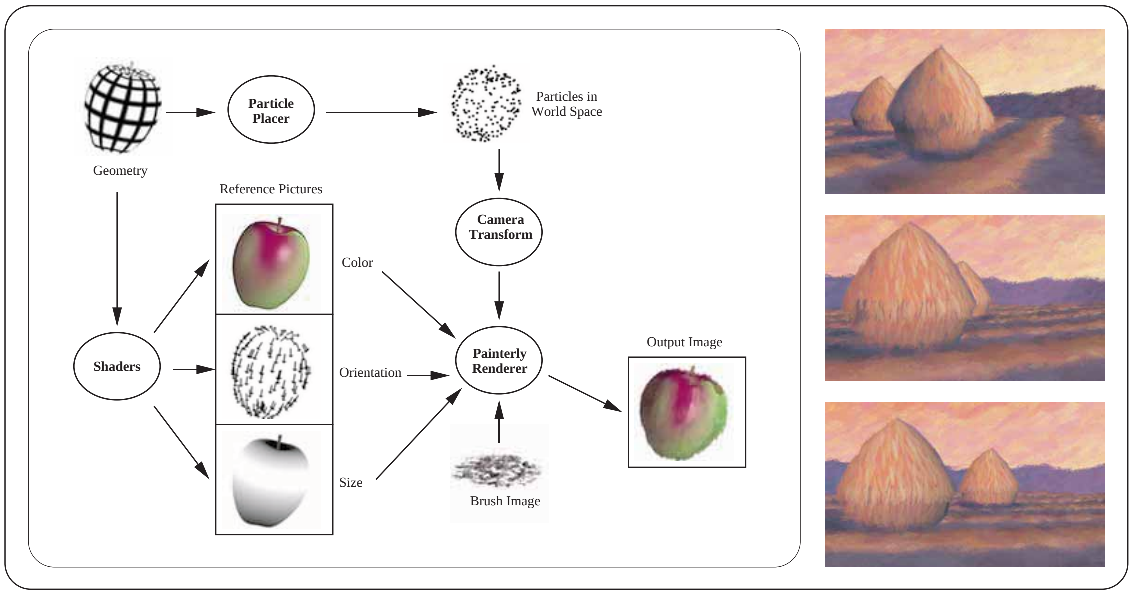


Figure 2.10: *Meier [36] Figure (a) shows the overall process of Meier's system. Figure (b) shows a sample rendered sequence.*

Markosian *et al.* [35] are the first to introduce major work in the area of real-time non-photorealistic rendering of 3D geometric objects. Underlying their system is a highly optimized version of Appel's hidden-line algorithm [1]. Their optimizations include the following: 1) the use of a rapid, probabilistic silhouette edge[5] finding method, 2) the exploitation of inter-frame coherence to speed the process of locating silhouette edges, and 3) the use of an improved and simplified version of Appel's hidden-line algorithm. Once the hidden-line algorithm has been run, the resulting edges are then rendered in various styles to produce different effects. The edges can be displayed directly to simulate a technical illustration

[5]Markosian defines silhouette edges as edges that are adjacent to both front and back facing polygons. Finding these edges is the first step in Appel's hidden line algorithm.

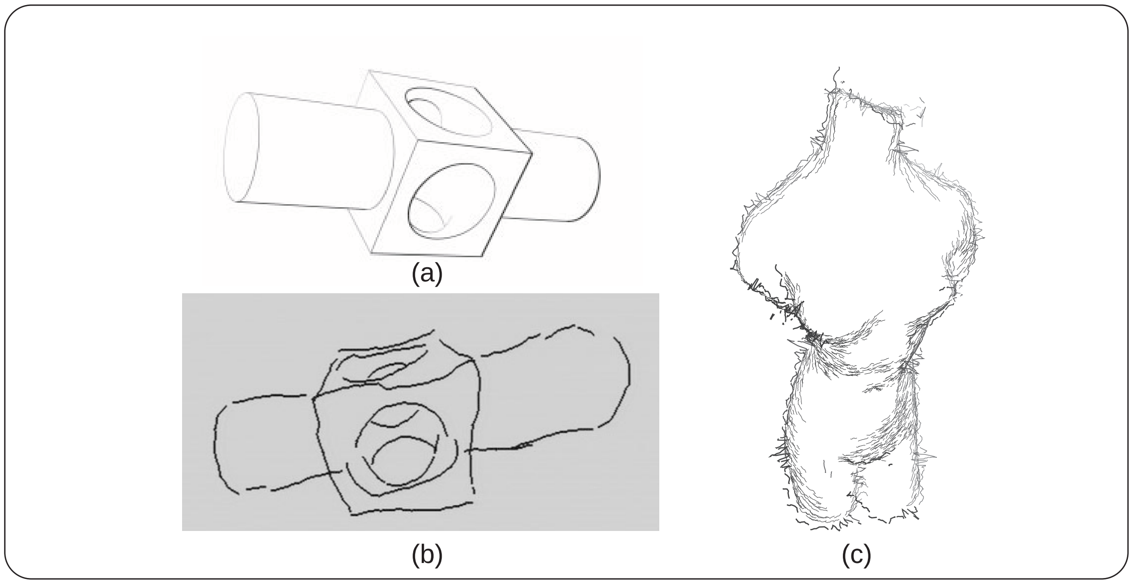


Figure 2.11: *Markosian et al. [35] Figure (a) shows a hidden-line drawing. Figure (b) shows the same model rendered non-photorealisticly. Figure (c) shows a more complex model rendered non-photorealisticly with shading.*

style, or perturbed to create a sketchy image. The edges can also be traced with texture-mapped strokes. Finally, Markosian discusses a method for generating shaded images with the simplifying assumption that the light is located at the camera position. Shaded strokes are placed on the surface of the object and are drawn only when the surface normal and the direction to the camera are greater than some predetermined degree. Results of Markosian's renderer are shown in Figure 2.11.

Coutts *et al.* [7] introduces a method for generating pen-and-ink style illustrations using streamlines. Coutts' technique begins with a 3D geometric model from which hidden lines are found. Six two-dimensional vector fields are also calculated which are later used for guiding streamlines. These vector fields are defined by various directions of flow across the objects in the scene to be rendered. Coutts also uses a tone map, generated by ray-tracing the underlying geometry, to determine tone and stroke spacing. Lastly, Coutts adds an ID map for identifying objects in the image plane. This ID map allows different objects to be rendered with different parameters. Once the flow fields are generated, streamlines, which simulate pen strokes, can be created. Furthermore, streamlines from various vector fields can be composited to create various cross-hatching type effects. To generate the streamlines, Coutts introduces a novel algorithm that grows streamlines from seeds on the image plane. In the initial case of a blank image, random seeds are sprinkled around the image. Then, as streamlines are grown, new seeds are placed at a prespecified distance to either side of it to ensure that streamlines are evenly spaced. A visit mask is also added that is updated as each streamline is drawn. The visit mask is an image buffer that records the regions of the drawing covered by the new streamline. To ensure that the correct tone in the final image is achieved, the width of the streamline drawn in the visit mask is widened for lighter regions. Streamlines that grow into regions marked visited are clipped. A sample of Coutts' output is shown in Figure 2.12.

Finally, Gooch *et al.* [17] introduce a new shading model for the automatic generation of technical illustrations. Their model, in contrast to a standard diffuse lighting model, drops the ambient lighting term, and instead, interpolates between two colors depending on the surface normal orientation to the light source. Points on the surface that are oriented toward the light are shaded with a warm color, k_{warm}. Points facing away from the light source are shaded with a cool color, k_{cool}.

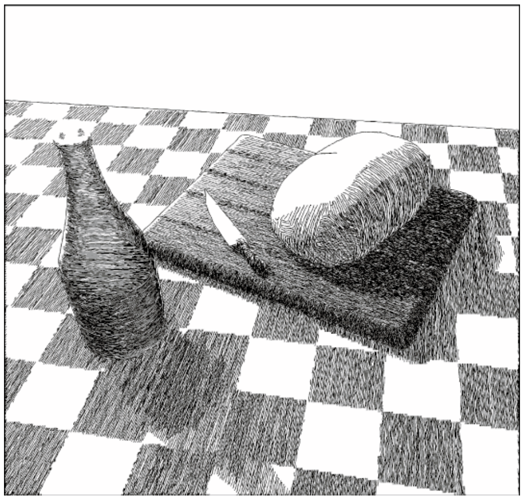


Figure 2.12: *Coutts et al. [7] A sample image generated from Coutts' pen-and-ink illustration system.*

Special care is taken to chose k_{warm} and k_{cool} so that they are similar to those colors used by technical illustrators. While rendering the final image, edge lines are also drawn to produce the final technical illustration effect. Lastly, Gooch introduces a technique to generate illustrations of anisotropic metallic objects. This is done by simply applying randomly generated grey lines around an object. The line closest to the light source is forced to white. The remaining lines are then interpolated between these intensity values and the resulting grading is blended with the lighting model to create the effect of metal shading. Results are shown in 2.13.

2.1.3 Natural Media Simulation

A few authors have researched methods focused primarily on the simulation of natural media. With the exception of the work done by Curtis *et al.*, these techniques are generally not complete non-photorealistic rendering systems because they only

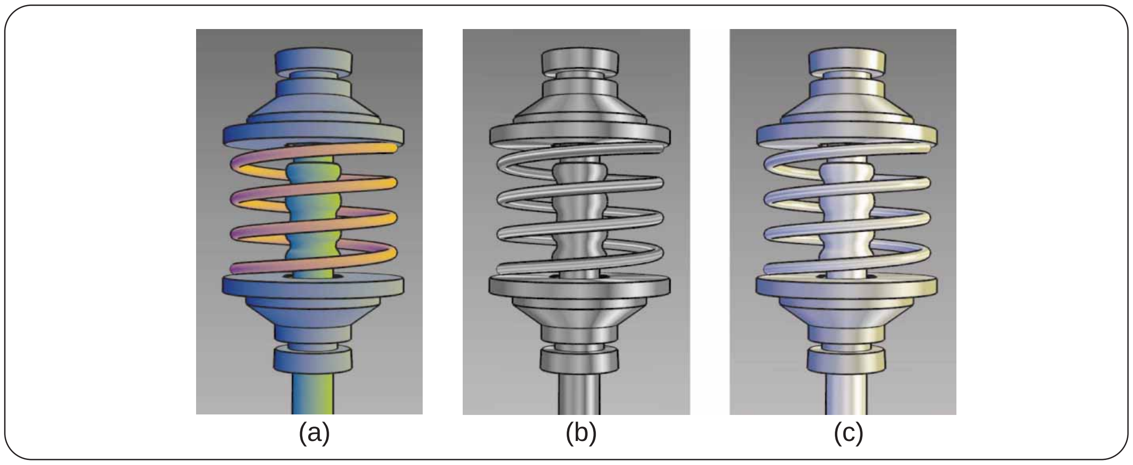


Figure 2.13: *Gooch et al. [17] Figure (a) shows a machined part rendered with Gooch's new lighting model. Figure (b) shows an illustration of an anisotropic metal part. Figure (c) shows Gooch's lighting model blended with the metal shading model.*

simulate media and do not have the capability to generate NPR images. However, natural media simulation techniques are a critical component of any NPR system and will be briefly reviewed in the following section.

Strassman [51] introduced a method to simulate brushes by accurately modeling the way a wet brush works. Bleser *et al.* [3] describes a charcoal drawing system which utilizes a look-up table of sample charcoal strokes. A digitizer's orientation and pressure are used to index into a table to find a charcoal texture with which to draw. Velho *et al.* [53] introduced a new method for halftoning images in which they use space filling curves instead of standard dithering algorithms. One surprising artifact they achieve on top of a new halftoning algorithm is the feel of charcoal shading on rough paper. Hsu *et al.* [22] introduced *skeletal strokes.* They described a method for stretching bitmaps along arbitrary splines and/or polylines to simulate painterly style strokes. One advantage of their system for interactive painting programs is the ease of modifing strokes by simply editing their underlying curve. Lastly, Curtis *et al.* [8] introduced a system for simulating water color paint (Figure 2.14a). To do this, they identified key components of watercolor and its interaction with paper, and then performed a fluid simulation given these components. In addition to an interactive painting program, Curtis also defined methods for the automatic water colorization of both photographs and 3D geometry. For photographs, Curtis requires the user to first segment the image into regions. Then, for each region, the user must choose pigments for the system to paint with. The system then separates each region into its component pigment colors. Finally, the system iteratively brushes over the image, adding water where the image is too dark and adding pigment where the color does not match correctly. For 3D scenes, the geometry is rendered to generate a source image. The remainder of the process is similar to that for photographs, except that the user is not required to manually separate the rendered scene into regions because this information already exists in the underlying model. Lastly, watercolor animations can be created by simply applying the process to successive frames of a 3D animated sequence. Figure 2.14b shows a sample photograph converted into a water color painting.

Finally, the commercial industry has also done much work in simulating natural media. MetaCreation's Painter [37] software provides excellent tools for simulating various natural media including, charcoal, pens, pencils, and paints. Futhermore, Adobe Photoshop [24] provides image filters which convert images into various natural media style drawings including charcoal and pencil.

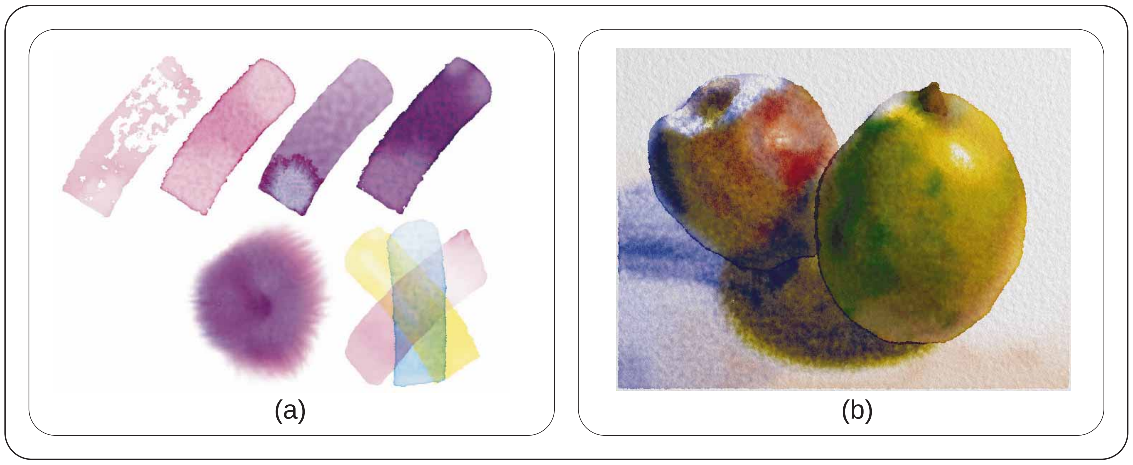


Figure 2.14: *Curtis et al. [8] Figure (a) shows the effects that can be simulated by Curtis' system. Figure (b) shows the results of water colorizing a photograph.*

2.2 Image Processing and Machine Vision

The goal of this thesis is to produce artistic charcoal-style renderings from portrait photographs. However, extracting facial information from a photograph is still an unsolved problem and remains an active research area in computer vision. Even with a well segmented image, it is still difficult to extract from a photograph the geometry information needed to apply some of the NPR methods described above.

Ultimately, the generation of artistic images from portrait photographs is an exercise in image processing and computer vision. Although we do not provide a completely automated solution to the generation of our renderings, we do use many image processing techniques to provide tools with which the user can coach the rendering process to generate the final results. In general, *Machine Vision* by Jain *et al.* [26] and *Digital Image Processing* by Gonzalez *et al.* [16] are good sources for general information in the fields of machine vision and image processing.

In the following sections we focus on the related work in three of the major areas that we draw from in our technique. In Section 2.2.1, we look at previous work done in the field of Hough transform research. In Section 2.2.2, we briefly review the field of image segmentation techiques, and in Section 2.2.3, we explore research done in the area of facial feature detection.

2.2.1 The Hough Transform and Orientation Filter

One step in generating the final artistic rendering is to extract a local orientation field from hair regions in the source portrait. We can then use this orientation field to draw charcoal-style streamlines to simulate the sketching of hair.

Although we introduce a novel use of the Hough Transform to extract local orientation from the source image, our technique is not the only method used to accomplish this extraction process. Picard *et al.* [39] provides a short survey of other methods for computing local orientation in their work. Also, Jähne [25], in his book, presents a detailed discussion of the local orientation problem as well as providing details of a few previous approaches.

Picard notes that most common methods of extracting orientation information are done by applying a small set of filters at pre-specified angles and scales. Each of these filters then reports a response for its particular direction. The responses of all the filters are then averaged to find the general orientation of the region. Knutsson's work *et al.* [28] on the Quadrature Filter Set Method is an example of an approach based on this method. This method will be discussed in Chapter 4 in order to compare a more common approach to the technique that we introduce.

Finally, Picard states that other methods that extract orientation directly have been explored using local derivatives, moments in the spatial and Fourier domains, and the Fourier spectrum directly.

In order to extract orientation from a source image, we apply a series of tiny Hough Transforms to the small region surrounding each pixel in the source image. Although the details of the transform and the orientation filter will be discussed later in more detail in Chapter 4, we will now cite some of the relevant work from the Hough transform research community.

The Hough transform was introduced by Paul Hough [21] in 1962 as a method for extracting curves in bubble chamber photographs. In our application of the Hough transform, we use a common Hough transform variant presented by Duda and Hart [13] which uses a (ρ, θ) parameterization of lines to avoid problems with infinite slopes.

In general, there are two excellent reviews of research related to the Hough transform. In 1988, Illingworth and Kittler [23] presented "A Survey of the Hough Transform" and in 1993, Leavers [31] presented his paper, "Which Hough Transform?"

Similar work to our orientation filter has been done by Bulot *et al.* [4]. Although Bulot's goal is to reconstruct contour lines from a source image rather than finding general local orientation, they similarily apply a series of local Hough transforms to each edge-detected pixel in the source image. However, rather than just finding general orientation, they also find the curvature of the edge. This information is then used to reconstruct a set of arcs that create a clean version of the original contour image. Unfortunately, Bulot's work is not directly applicable to finding hair details because their technique requires relatively equally spaced and clean contour images.

Speed is a large concern in our application of the Hough transform. Because it is not uncommon for portraits to be composed of hundreds of thousands of pixels, each application of the Hough transform must be fast in order to generate reasonable running times. One standard method of reducing the amount of computation required by the Hough transform is to use look-up tables [50].

Other optimizations to speed the computation time can be applied as well. Ballard [2] describes a method for reducing the dimension of the data accumulated in the parameter space of the Hough transform for analytic curves by taking advantage of derivative information. For the standard Hough transform, this optimization would reduce accumulating lines or sinusoidal curves in Hough space to accumulating only points, reducing the total amount of computation required. Davies [10] proposes a similar algorithm that uses a *foot-of-normal* parameterization derived from the gradient information. The foot-of-normal parameterization characterizes a line by the point on it that lies closest to the origin. This parameterization is valuable not only because lines are parameterized as single points in accumulator space, but also because it can be performed in the same xy space as the source image.

Other techniques can be used to decrease the computational work required by the Hough transform as well. Xu *et al.* introduced the Randomized Hough Transform (RHT) [57] for detecting lines. The RHT works by taking two random points in image space to generate one point in parameter space. Kiryati introduced the Probabilistic Hough Transform [27] that processes only a subset of all the points in the source image to reduce the total computation of the transform. Kiryati showed that only 5 to 15 percent of the source image pixels need to be processed for reasonable results.

Several hierarchical variations of the Hough transforms have been presented that, if applied to our orientation filter, could provide faster computation. Futhermore, much work has been done in the area of accelerating the Hough transform through the use of advanced hardware architectures such as via SIMD machines. Although which of these methods may speed the computation of the orientation filter is questionable, they are viable options. Both the hierarchical techniques as well as the hardware accelerated techniques are well referenced in the Hough transform surveys presented by Illingworth [23] and Leavers [31].

2.2.2 Image Segmentation

The facial-feature shading algorithm we propose in this thesis depends on the ability to segment individual facial features in our source photograph. Once segmented, the individual components of each feature can then be drawn. The following section will review the general field of image segmentation, and then discuss the specific work that we chose to use in this thesis.

In their review of current image segmentation techniques, Pal and Pal [38] state that "hundreds of segmentation techniques are present in the literature, but there is no single method which can be considered good for all images, nor are all methods equally good for a particular type of image.” Furthermore, Pal and Pal note that “even the selection of an appropriate [image segmentation] technique for a specific type of image is a difficult problem.”

For this thesis, we use a simple segmentation algorithm which involves *grey level thresholding*. Grey level thresholding is a class of image segmentation algorithms in which an image is broken up into regions depending on the intensity value of each pixel. For simple images, such as a picture of a white box on a black background, these techniques work well. However, because this class of techniques generally rely on image histogram analysis, they tend to ignore special details which can lead to incorrect segmentations. The basic philosophy behind grey level thresholding is that pixels of similar grey values should be grouped together. However this assumption may not be true on many occasions. Noisy images are a good example of when this assumption is not valid.

We choose to use a simplified version of the segmentation algorithm introduced by Lim *et al.* [33] to threshold the facial features in our source photograph. Although the segmentation algorithm possesses the limitations described above, it produces good results in our application. Details of this algorithm will be discussed in Chapter 6. At the core of the algorithm, we employ the concept of *scale space filtering*. More about this topic can be found in [52].

2.2.3 Facial Feature Recognition

In our proposed NPR system, we require the use of facial feature recognition to isolate and identify individual parts of the face. Left and right eyebrows, for example, are both isolated and hatched independently. Eyes are handled similarly, except different shading parameters are applied.

The general face recognition problem is to match an image of one's face to a corresponding record stored in a database. Although applications of this research have mainly been in law enforcement (i.e. matching photos to those in mug-shot albums), new applications of user authentication in secure systems have arisen. Examples of such systems include automatic teller machines, computer systems, and credit card verification.

In their survey of face recognition, Chellappa *et al.* [6] break the recognition problem into three main categories, *segmentation*, *feature extraction*, and *recognition*. Segmentation involves isolating facial regions in photographs. Feature extraction involves finding various features in the face. Lastly, the recognition phase takes the features and uses them to match faces stored in a database. Samal *et al.* [46] have also published an excellent review of facial feature recognition.

For our work, we are primarily concerned with the feature extraction phase of face recognition. Segmentation is currently done manually by the user. Because we are only generating an artistic rendering, recognition is not necessary.

Although *feature extraction* is only a subproblem of face recognition, many solutions to this problem have been proposed. For our system, we take one general approach of searching for darker regions in the facial region and then classifying them based on their relative location to each other. Other methods for feature extraction can be found in the surveys by Chellappa *et al.* [6] and Samal *et al.* [46].

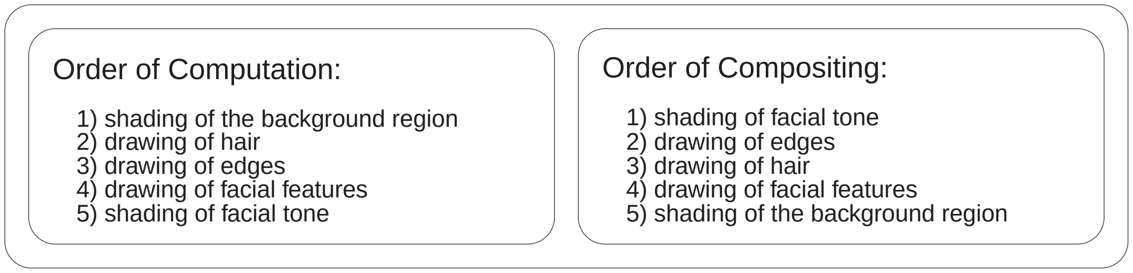
Chapter 3

System Overview

While creating a system for generating charcoal sketches from portrait photographs, we defined a rendering pipeline that consists of the following five major steps: rendering of facial-features, hair, edges, facial-tone, and background-shading. Each of these major steps is further composed of a series of sub-steps or operations. These operations include filters, such as a Gaussian blur, or drawing operators, like the hair rendering algorithm. The following chapter presents a high-level overview of the major-steps and operations that constitute the artistic rendering process (Section 3.1). Also, because the artistic rendering process is interactive and closely tied to the artistic rendering application, a brief overview of the software developed to implement the rendering process will be described (Section 3.2).

3.1 Artistic Rendering Process

Each of the five major steps in the artistic rendering process is responsible for generating a portion of the final artistic rendering. Although most of the steps are computationally independent, a few dependencies do exist. Thus, we have defined



Order of Computation:

1) shading of the background region
2) drawing of hair
3) drawing of edges
4) drawing of facial features
5) shading of facial tone

Order of Compositing:

1) shading of facial tone
2) drawing of edges
3) drawing of hair
4) drawing of facial features
5) shading of the background region

Figure 3.1: *A comparison of the computation versus the compositing process.*

the computational ordering of the steps in the artistic rendering process to be 1) the shading of the background region, 2) the drawing of hair, 3) the drawing of edges, 4) the drawing of facial features, and 5) the shading of facial tone. However, this ordering differs from the ordering in which the results from each step are composited into a final image. For example, the background region is the first to be processed but the last to be drawn. Thus, for the compositing process, a loose ordering is again implied. We have defined the ordering of drawing to be the following: 1) the shading of facial tone, 2) the drawing of edges, 3) the drawing of hair, 4) the drawing of facial features, and 5) the shading of the background region. Figure 3.1 compares the ordering of the computation and compositing processess. Figure 3.2 illustrates how the results of each of the five major steps are composited into a final artistic rendering.

3.1.1 Background Shading

The background region of the source photograph must first be segmented for the background shading step (Figure 3.6b). This user-interactive separation process is relatively straight-forward because our artistic rendering system requires pho-

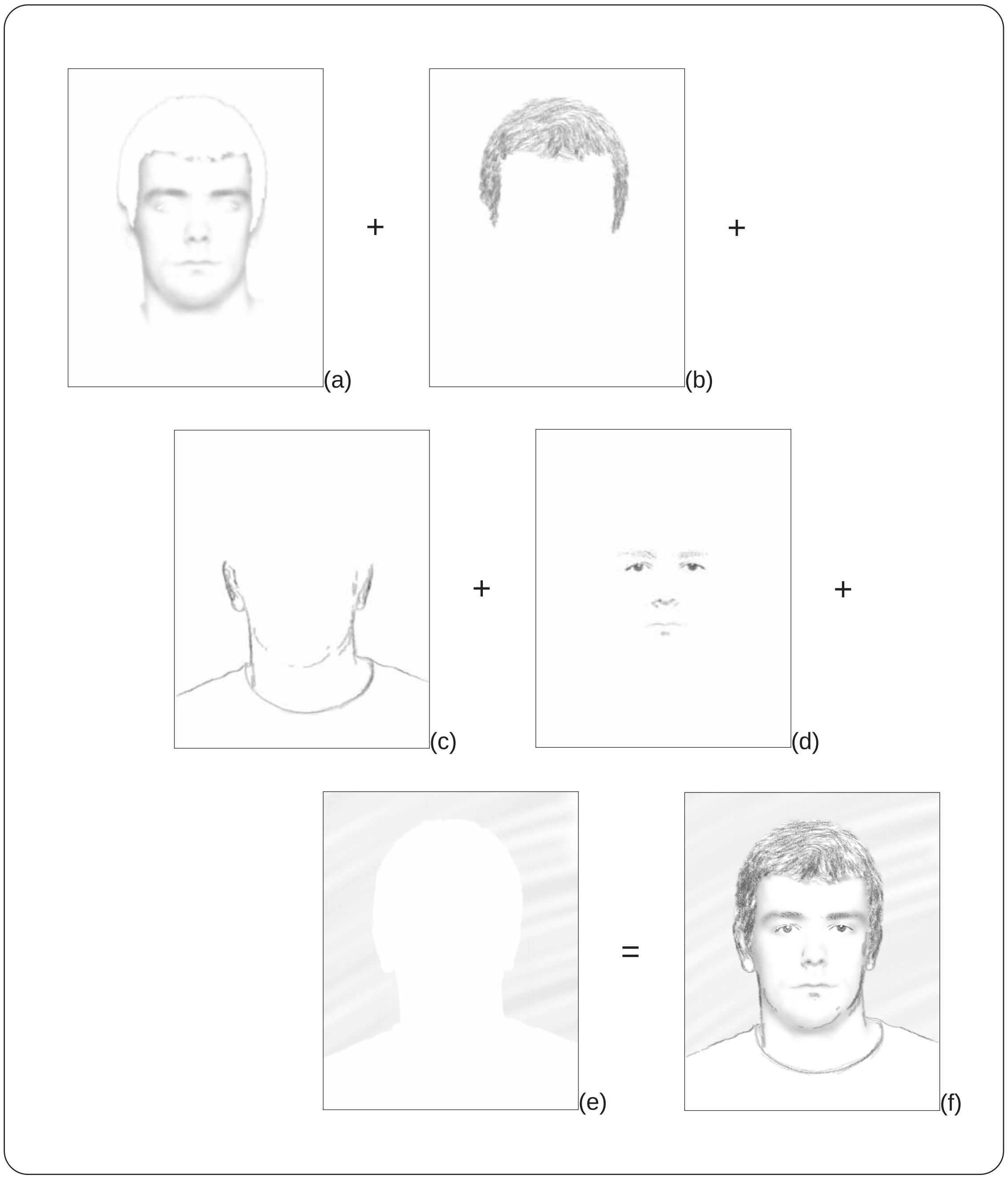


Figure 3.2: *An overview of the artistic rendering process. The (a) facial tone, (b) edges, (c) hair, (d) facial features, and (e) background region are compositied for form (f) the final artistic rendering.*

tographs to be taken with a blue-screen. Once isolated, the background can then be rendered artistically. Figure 3.6c shows an example of an artistically shaded background region.

Because later steps in the artistic rendering process require knowledge of which pixels compose the background area, this region must be temporarily stored. We define this storage buffer as the *image mask*. In general, some steps require the image mask to remove regions in the image that have already been visited by previous steps.

3.1.2 Hair Drawing

Once the background region has been isolated, the hair region can then be processed. In general, we define the hair region to be any hair in the portrait excluding the eyebrows. Although the eyebrows are also technically hair regions, they will be handled specially by the facial feature drawing algorithm.

The first operation in this step is to isolate the regions in the image that represent hair (Figure 3.7b). As with the selection of the background region, this separation process is a user-interactive one. Once the hair region has been isolated, the image is then converted to grey-scale (Figure 3.7c). The grey-scale conversion uses the definition of CIE luminance [41],

$$Y_{709} = 0.2125R + 0.7154G + 0.0721B$$

The grey-scale image is then smoothed with a Gaussian blur (Figure 3.7d). Our experimental results show that a standard deviation of 2 for the Gaussian kernel generally produces good results. Next, a standard line-detection algorithm [26] is applied to the smoothed image (Figure 3.7e).

The Hough orientation filter is then applied to the line-detected image (Figure 3.7f). In short, the orientation filter computes the local orientation of the lines in the image for each point in the image. Because of noise, quantization errors, and the inherent fact that hair tends to be somewhat messy, the computed orientation field must be smoothed before it can be used as a basis for the hair-drawing algorithm (Figure 3.7g). To smooth the orientation field, an adapted version of a Gaussian blur for orientation fields is applied. From experimental results, we have found that a large standard deviation of 15 is necessary to adequately smooth the hair region while preserving the general hair orientation. With the resulting orientation field, the hair drawing operation can then be applied (Figure 3.7h). Finally, as with the background region, the hair region must be added to the image mask.

3.1.3 Edge Drawing

After the hair region is processed, the edges that are to be drawn in the final image must be extracted. The edge drawing process is the most user-interactive step of the artistic rendering pipeline because it requires the user to manually identify which edges in the image are to be kept. The identification process is done by erasing the unwanted edges from the edge-detected image. Although seemingly tedious, the edge removal process is actually quite straight forward. The use of the image mask and a de-speckling operator help remove unwanted edges in the hair region and false edges created by noise. Furthermore, a loose definition of which edges need to be kept also simplifies this process. Small edge segments that lie close to wanted edges and that are hard to remove can be left alone without drastically changing the resulting drawing. This operation usually takes three to five minutes.

The first operation in the edge drawing step is to find edges in the source photograph. Although any edge operator can be used, we choose to use the Canny edge detector [5]. Canny's edge detector begins by smoothing a source image with a Gaussian blur (Figure 3.8b). Next, the gradient of the blurred image is computed, and then a non-maxima suppression operator is applied to the gradient vector field. The resulting pixels from these operations form the edges of the image (Figure 3.8c). The image mask, which at this point contains the background and hair regions, is subtracted from the edge-detected image to assist in the removal of unwanted edges (Figure 3.8d). In some cases, the edge between the hair region and the background region will not be eliminated by this operation. This is because the user-selected background and hair regions are not guaranteed to be abutting. The edge that lies between the two regions, since it is not contained in either one, will remain after the subtraction operation. To further eliminate unwanted edges, a de-speckling algorithm removes line segments that fall below a user-specified length (Figure 3.8e). In general, a threshold length of 10-15 works well.

The next step in the edge drawing step requires the user to manually remove unwanted lines from the remaining edges in the edge-detected image (Figure 3.8f). Although highly user-interactive, the removal process is relatively straight-forward and simple. In general, the determination of which lines are to be kept for the final drawing is entirely a subjective process, and the user is free to determine which lines in the edge detected image are to be kept. We found that isolating the lines that represent the chin, ears, neck and shoulders is generally a good metric. The detailed lines that compose the facial features should usually be removed to produce good results. In addition to these general guidelines, the lines that are kept for the drawing process do not need to be precisely isolated from the original edge-detected image. This unnecessary need for precision is due to the general sketchy nature of our target charcoal drawing style.

Finally, the edge information can be rendered in a charcoal style. Figure 3.8g shows the final result.

3.1.4 Facial Feature Drawing

The ability to find facial features (i.e. the eyebrows, eyes, nose, and mouth) in the source image lies at the heart of the facial feature drawing step. Our algorithm is based on the assumption that darker regions on the face are significant markers for the locations of facial features. This assumption works well for portrait photographs because the controlled lighting frees us from the need to worry about shadows. However, the algorithm does make some limiting assumptions. Primarily, faces with darker skin tone or dark birth marks will cause problems. Also, occluding objects, such as glasses, will mislead the feature finding algorithm as well.

The first operation in the drawing of facial features requires the user to select the region of the source photograph that represents the face. However, unlike previous selections, this selection process can be relatively imprecise. All that the user must do is to include all of the facial features in the selection while excluding regions outside the face itself. Figure 3.9b shows a sample of a selected region.

Once selected, the image mask must be subtracted from the image. This operation is necessary when the subject in the source photograph has facial hair such as a mustache or beard. Because facial hair should have been selected in the hair drawing step, subtracting the image mask should remove any hair regions from the facial region. As mentioned previously, eyebrows are an exception to this operation because they are considered by our system as a facial feature and not as a hair region.

Next, the lighting in the image must be normalized (Figure 3.9c). This interactive process requires the user to pick various points on skin regions across the face. This lighting normalization is based on the observation that faces are somewhat cylindrical in shape and that the lighting in a portrait studio tends to primarily come from the direction of the camera. In the original photograph, the sides of the face tend to be much darker then the center. After the normalization process, the tone across the face is fairly regular.

From this normalized image, a grey-scale image is computed with the same method described earlier (Figure 3.9d). The brightness and contrast of the image are then increased (Figure 3.9e). This increase helps the thresholding process by washing away any of the remaining shadows of the source image.

Next, the features in the face can be found by a clustering algorithm and prior knowledge of where features lie in relation to one another. Figure 3.9f shows the results of this clustering process. The different colors in this image represent different facial features that were found. With the clustered image, the individual facial features can then be drawn (Figure 3.9g). Finally, the regions of the portrait that represent the eyes must be added to the image mask. Note, however, that we do not add the entire region represented by the facial features to the image mask. The reason for this exception is mainly aesthetic and will be explained in the Section 3.1.5.

3.1.5 Facial Tone Shading

The final step of the rendering process is the generation of facial tone from the source image[1]. Unlike the previous steps, the result of the facial tone shading step is a grey-scale image rather than a set of brush strokes. This image will be used as a backdrop on top of which the other elements of the sketch will be drawn.

The first operation in the facial tone shading step is to subtract the image mask from the source photograph (Figure 3.10b). Currently, the image mask contains the regions of the image that represent the background area, the hair, and the eyes. After subtracting, we are left with the regions of the image that represent the face (excluding the eyes), neck, and shoulders. The reason that only the eyes are excluded, rather than all of the facial features, is due to experimental results. In generating sketches, we found that by leaving the general tone of the images under the eyebrows, nose, and mouth unmodified, we were able to obtain a more pleasing final drawing. Subtracting these regions from the tone image often leaves

[1]Note that although we refer to this step as facial tone shading, it also applies to the shading of tone around the neck and shoulders.

discontinuities in tone around each feature. Figure 3.3a,b shows two images of an eyebrow. Figure 3.3a shows the eyebrow in the case where we subtracted the mask from the background tone. Figure 3.3b shows the same eyebrow in the case that it is not subtracted. Notice the unwanted halo-effect around the eyebrow in Figure 3.3a. This effect is due to the fact that it is difficult for the facial feature clustering algorithm to isolate the entire eyebrow region. The fuzzy nature of borders in natural images as well as the method in which we threshold the image before clustering are the source of this inability. The eyebrow in Figure 3.3b lacks the discontinuity and produces a better result.

Unlike the other facial features, the regions in the image that represent the eyes are subtracted from the tone image. Figure 3.3c,d illustrates why we chose to make this exception. Figure 3.3c shows the eye in the case where the eye region is subtracted from the background tone. Figure 3.3d shows the case where the eye region is not subtracted. Notice that when the eye is drawn over the underlying color from the tone map (Figure 3.3d), the highly specular regions of the eye are not visible.

After the subtraction operation, the image is converted to grey-scale (Figure 3.10c). The image is then heavily blurred (Figure 3.10d) and its brightness is strongly increased (Figure 3.10e). Note how this blurring process is limited to only the regions of the image that were not removed when the image mask was subtracted. The next step in the shading of facial tone is to then remove any masked regions in the image (Figure 3.10f). These regions are represented by black areas in the figure, and after the removal operation, they are replaced with

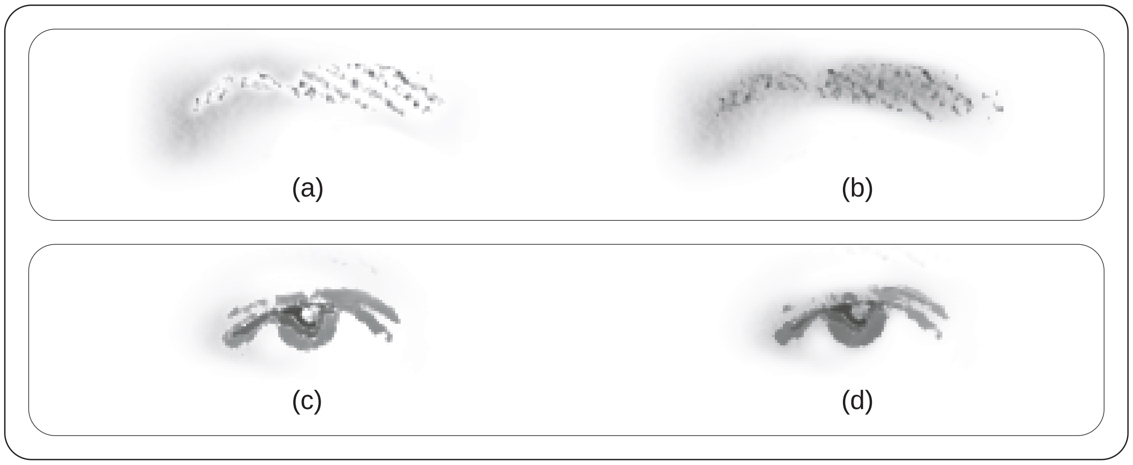


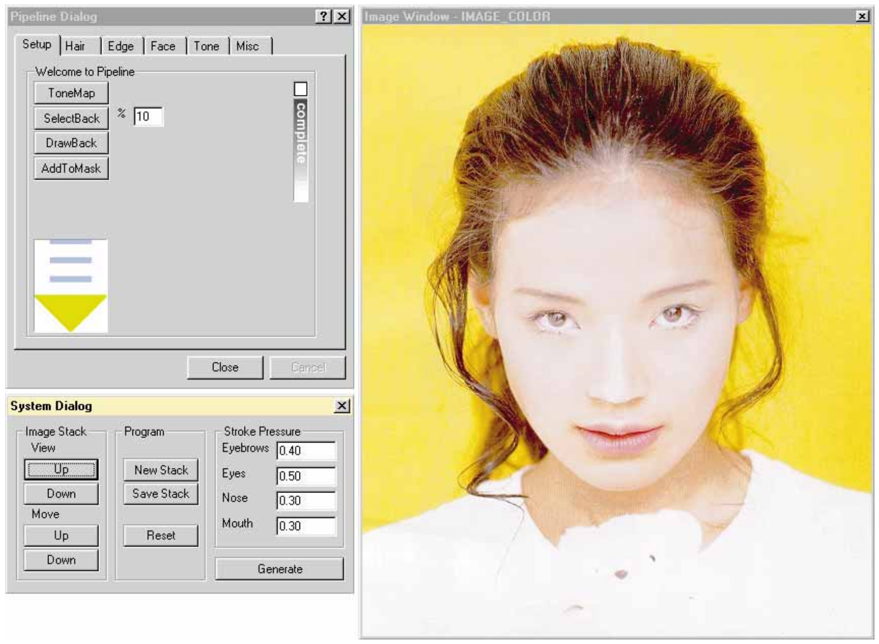
Figure 3.3: *A set of images illustrating the advantages of only including the eye regions in the image mask. Two eyebrows with the eyebrow regions included (a) and not included (b) in the image mask. Two eyes with the eye regions included (c) and not included (d) in the image mask.*

unmasked, white pixels. Next, the image is blurred again to smooth the rough edges that resulted from the image mask subtraction process (Figure 3.10g). The resulting image is a very smoothed and washed out version of the original image. Finally, the resulting image is blended with a paper texture to give the appearance of smudged charcoal (Figure 3.10h).

3.2 Application Architecture

The artistic rendering software provides a user-interface that closely parallels the underlying rendering pipeline. In general, to create an artistic rendering, the user begins by loading a source photograph and then proceeding down a linear progression of operations to produce a final image. Figure 3.4 shows a screen-shot of the interface.

After each operation in the series is applied, an intermediate image is gener-



% 10

Eyebrows 0.40
Eyes 0.50
Nose 0.30
Mouth 0.30

Figure 3.4: *A screen-shot of the user interface for the artistic rendering software.*

ated. This image is then used as a source image for the following operation. In our system, we loosely define the term *image* to describe either a grey-scale or RGB pixel array with some associated meta-data. This meta-data can include information such as a vector field, visibility mask, or pixel cluster information.

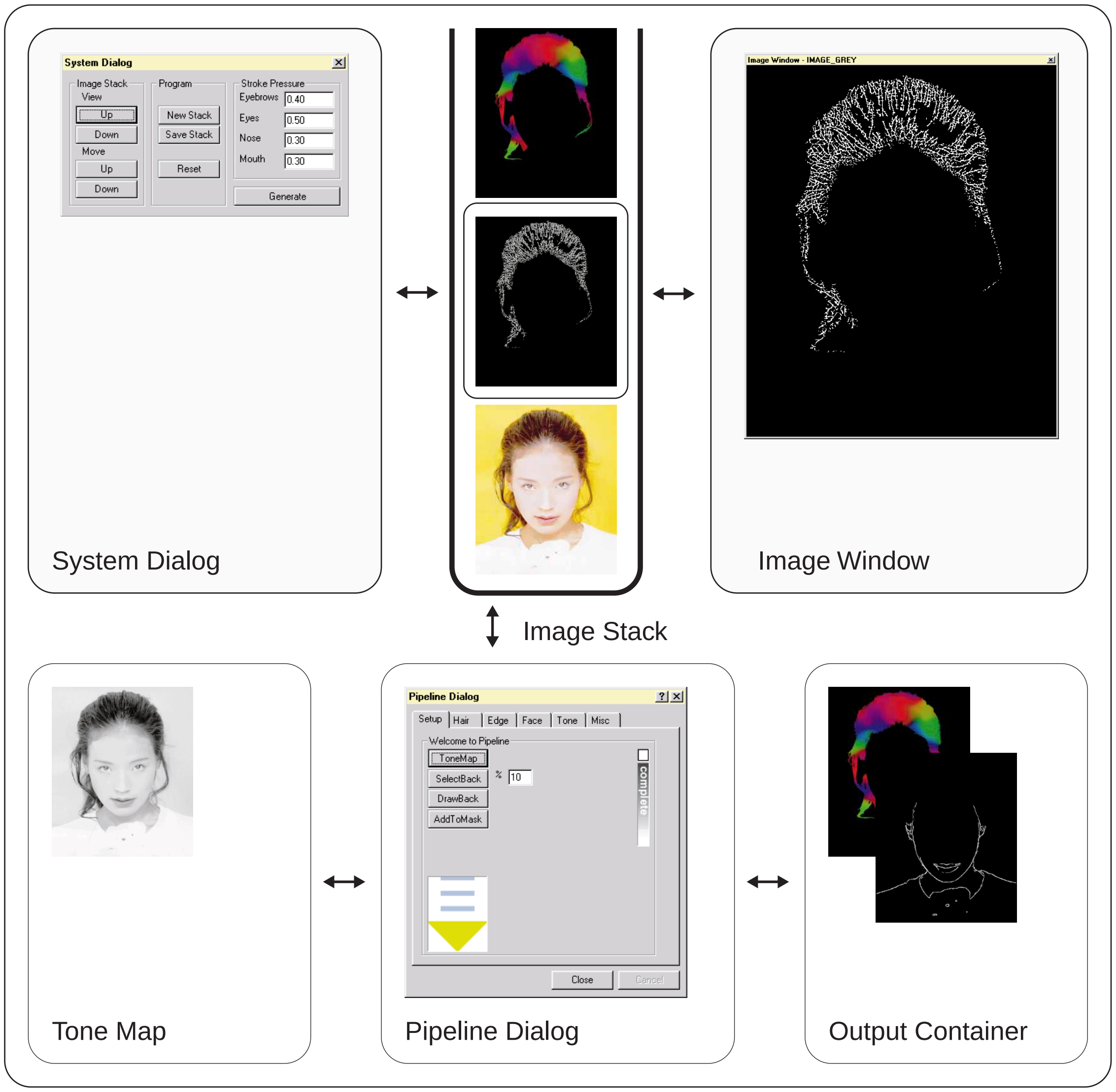
The central data structure to the artistic rendering software is the *image stack* (Figure 3.5). Operations use the top stack element as their source image when applied. Upon completion, the resulting images are pushed onto the stack to become the new top element. Through the System Dialog, the rendering software provides the ability to view and reorder previous images in the image stack. This functionality allows the user to reapply operations to previous images. The System Dialog also allows the user to generate the script that can later be imported into

MetaCreations' Painter software [37]. To generate this script, the user is allowed to choose the pressures of the brushes that will be used to draw the facial features.

In general, the user begins the artistic rendering process by loading a source photograph. Then, starting with the Setup Dialog (Figure 3.11), the user proceeds down a series of operations in the dialog box. When completed, a small notification box will turn red to signify that the current step is complete. At the completion of each step, the user is required to reset the image stack, reload the original image, and proceed to the next step. For example, when the user is done with the Hair rendering step, s/he is required to proceed to the Edge dialog. When all the steps are completed, the user can then generate the final artistic rendering.

In some cases, a pair of Cut and Paste operations will appear in the series of operations. Although we implement a few tools for image selection and painting, commercial applications such as Adobe Photoshop [24] generally do a much better job. Thus, we require an external image manipulation program to handle some of the user interactive selection and painting tasks required by our system. To interface with these external programs, we use the Win32 clipboard which allows the cutting and pasting of images between applications.

Following each of the major steps is a draw operation. This sub-step takes the top element of the stack and adds it to an *output container* (Figure 3.5). The output container stores copies of the image data that will later be used to generate the final artistic rendering. For example, if a vector field is needed for the hair generation algorithm, it will be stored in the output container regardless of whether it has been flushed from the image stack.



Eyebrows 0.40
Eyes 0.50
Nose 0.30
Mouth 0.30

% 10

Figure 3.5: *The structure of the artistic rendering software. At the heart of the rendering software is the* **Image Stack***. The* **System Dialog** *allows the user to view and reorder images in the stack. The* **Pipeline Dialog** *applies operations to the elements in the* **Image Stack** *to generate new elements. The* **Tone Map** *is used for the drawing of facial features. The* **Output Container** *holds images independently of the* **Image Stack** *until the user is ready to generate the final artistic rendering.*

Finally, an *image mask* and *tone map* are stored independently of the image stack (Figure 3.5). The image mask represents regions of the source photograph that have already been drawn or completed. In some steps, the user will be required to add the current selection to the mask. For example, after the user has isolated which pixels represent hair, s/he is required to add this region to the image mask. After the user has reset the image stack and reloaded the source photograph, the mask can then be subtracted from the image, leaving only those regions which have not yet been considered. The tone map is used only as a reference for the drawing of facial features.

Figures 3.11 to 3.15 step through the operations in the artistic rendering software.

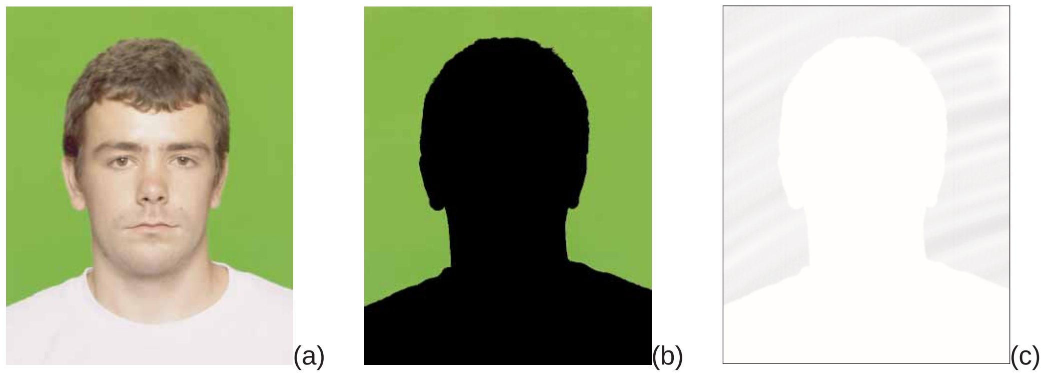


Figure 3.6: *A typical image stack for the rendering of the background region. (a) The original image. (b) The background region after it has been selected. (c) The final artistically rendered result.*

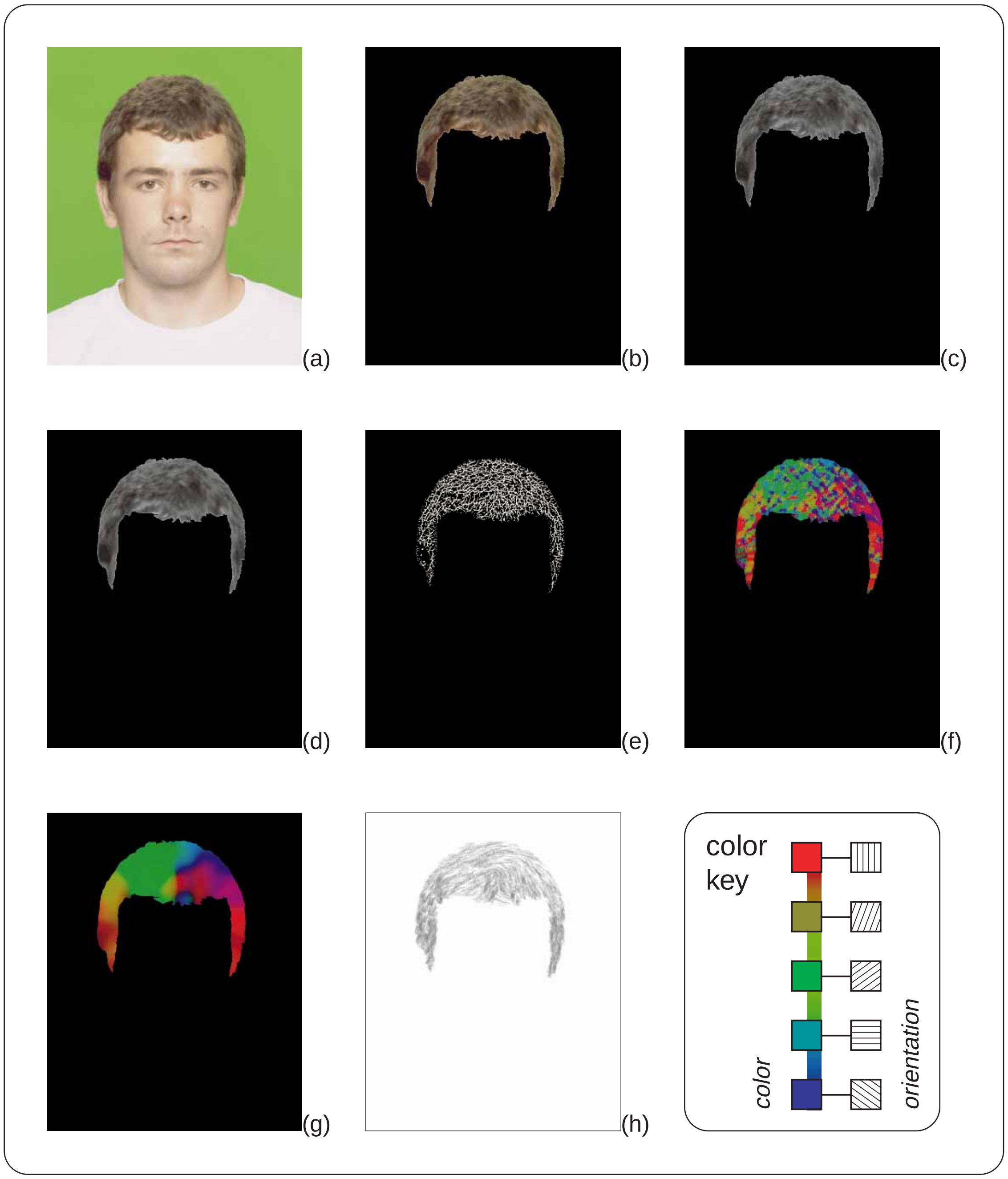


Figure 3.7: *A typical image stack for the rendering of hair. (a) The original image. (b) The hair region after it has been selected. (c) The hair region converted to grey-scale. (d) The hair-region blurred. (e) The line-detected version of the hair region. (f) The result of applying the orientation filter. (g) The smoothed orientation field. (h) The final artistically rendered result.*

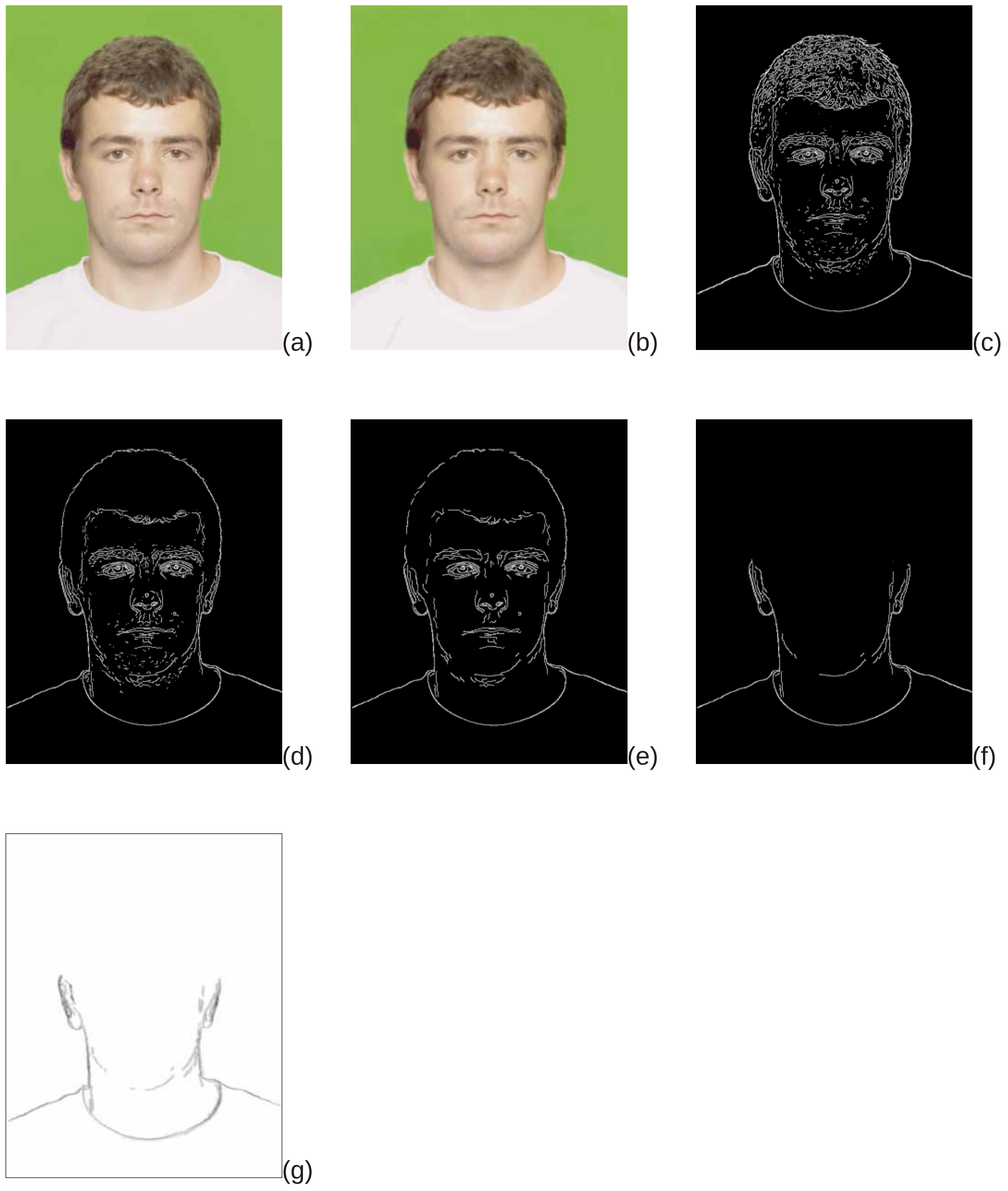


Figure 3.8: *A typical image stack for the rendering of edges. (a) The original image. (b) The image after blurring. (c) The edge-detected image. (d) The image after subtracting the image mask. (e) The image after despeckling. (f) The result after a user has removed unwanted edges. (g) The artistically rendered result.*

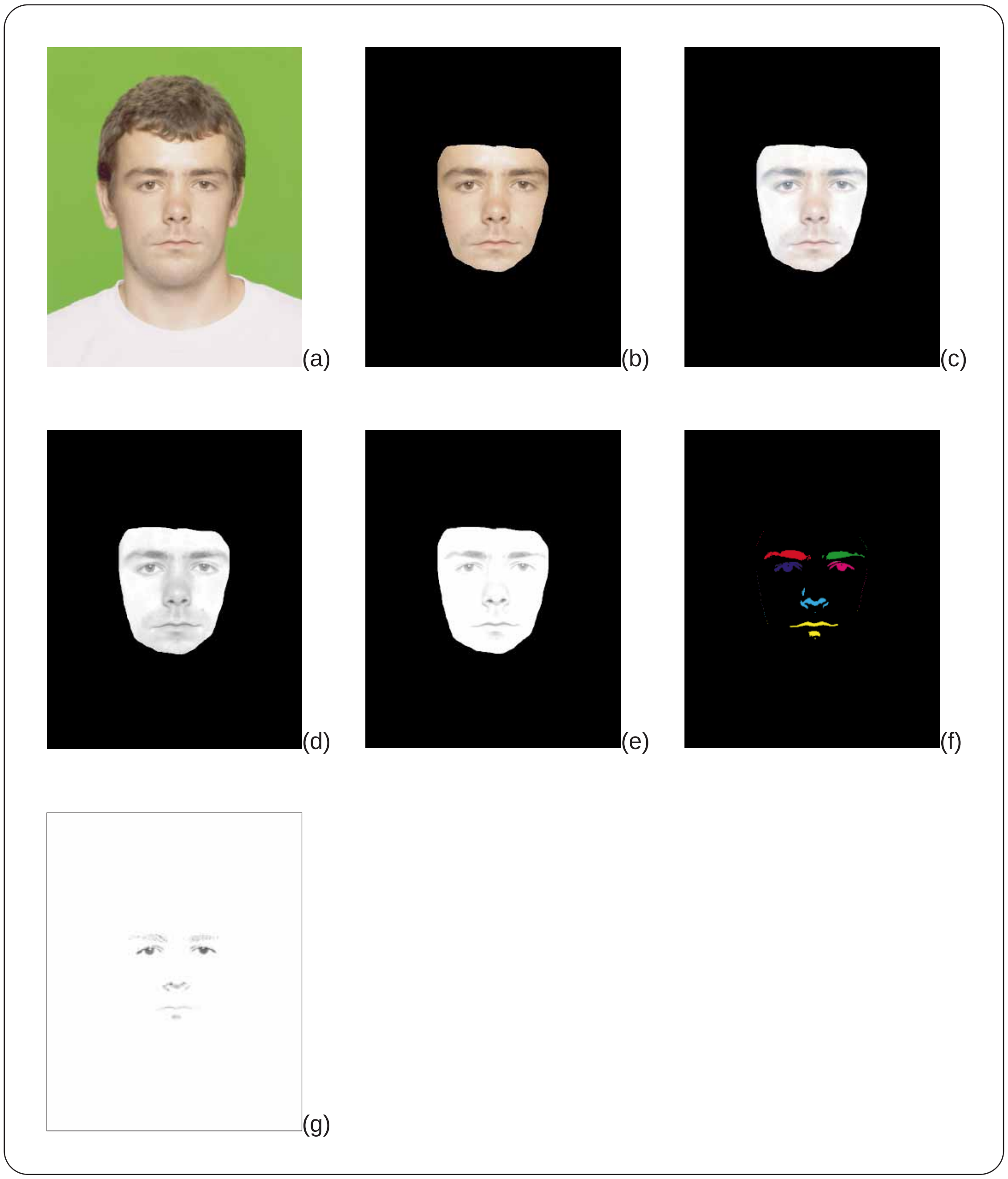


Figure 3.9: *A typical image stack for the rendering of the facial region. (a) The original image. (b) The facial region after being selected by the user and subtracting the image mask. (c) The result of the lighting normalization process. (d) The image after converting to grey-scale. (e) The image after increasing the brightness and contrast. (f) The image after the facial feature finding algorithm. (g) The final artistically rendered result.*

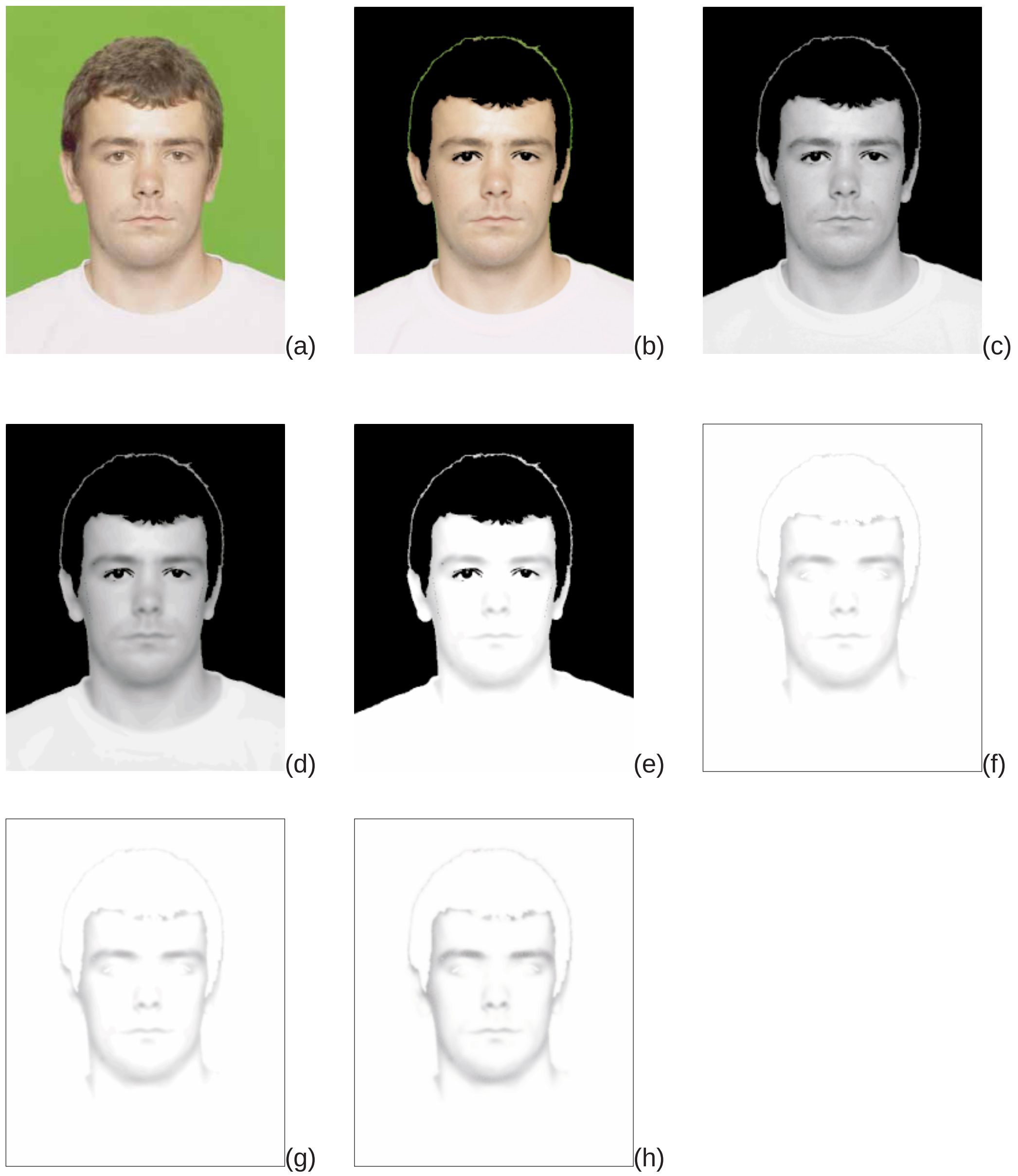
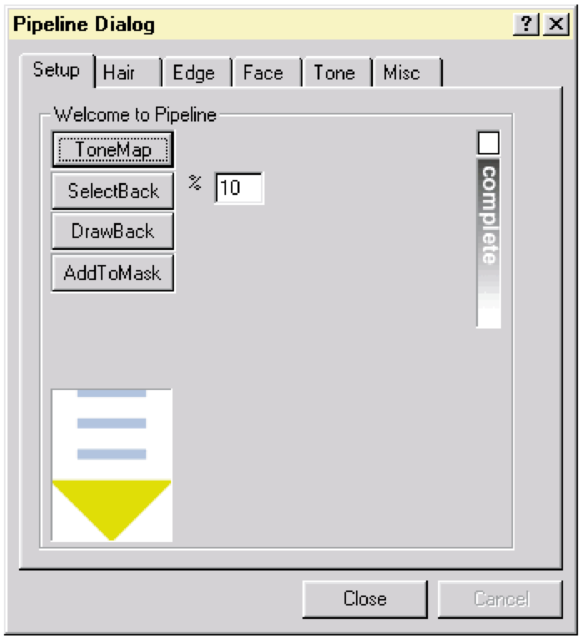


Figure 3.10: *A typical image stack for the rendering the tone in the image. (a) The original image. (b) The image after subtracting the image mask. (c) The image after converting to grey-scale. (d) The result after applying a blur. (e) The image after the brightness has been increased. (f) The image after removing the masked area created by the substraction process. (g) The image blurred again. (h) The final tonal shading.*



Description of Operations:

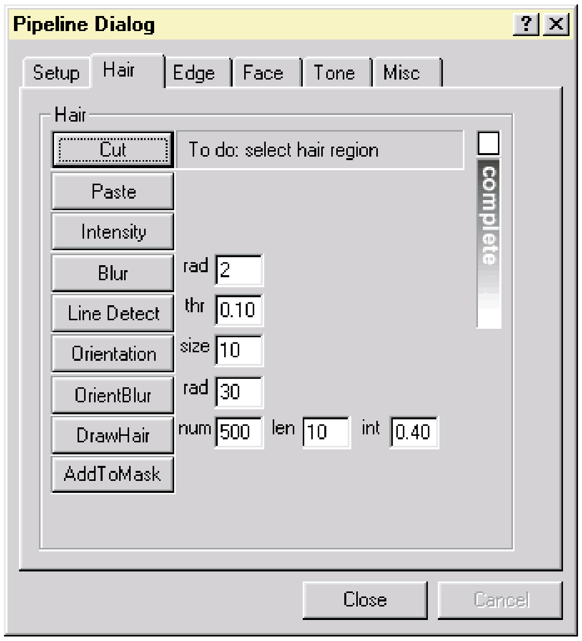
ToneMap: Sets the global *tone map* with the passed image.
input: `image_color`
output: <none>

SelectBack: Segments the background region of the image. User is required to identify a small rectangular sample of the background region to be selected.
input: `image_color`
output: `image_color`
parameters:
%: tolerance of pixels to be included in the selection

DrawBack: Sends the selected background region to the *output container*.
input: `image_color`
output: <none>

AddToMask: Adds the selected background region to the *image mask*.
input: `<any image>`
output: <none>

Figure 3.11: *The Setup dialog box.*



Description of Operations:

Cut/Paste: User is required to use an external program to select the hair region.
  input: `image_color` or `image_grey`
  output: `image_color`

Intensity: Converts a color image into a grey-scale image.
  input: `image_color`
  output: `image_grey`

Blur: Blurs the image.
  input: `image_color or image_grey`
  output: `image_color or image_grey`
    parameters:
  rad: radius of the blur

LineDetect: Detects lines in a blurred image.
  input: `image_grey`
  output: `image_grey`
    parameters:
  thr: minimum threshold with which to generate a responce.

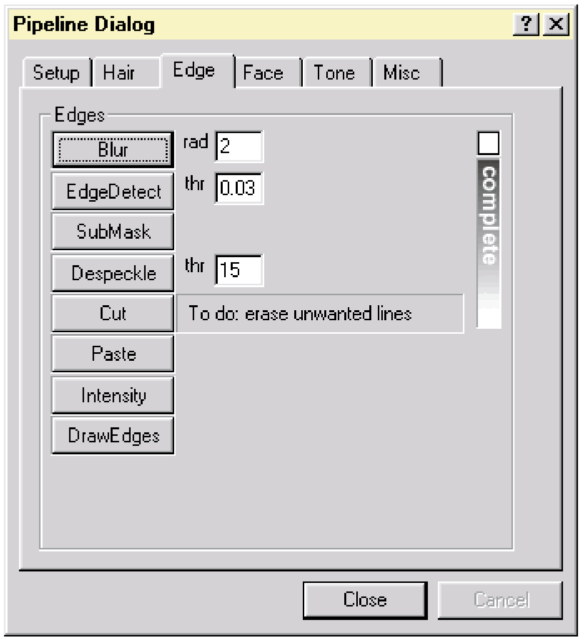
Orientation: Computes the orientation of lines in the image.
  input: `image_grey`
  output: `image_color_orient`
    parameters:
  size: the size of the orientation filter.

OrientBlur: Smooths the orientation vector field.
  input: `image_color_orient`
  output: `image_color_orient`
    parameters:
  rad: the size of the blurring filter.

DrawHair: Sends data to the *output container.*
  input: `image_color_orient`
  output: <none>

AddToMask: Adds the selected hair region to the *image mask*.
  input: `<any image>`
  output: <none>

Figure 3.12: *The Hair dialog box.*



rad 2
thr 0.03
thr 15
To do: erase unwanted lines

Description of Operations:

Blur: Blurs the image.
input: `image_color` or `image_grey`
output: `image_color` or `image_grey`
parameters:
rad: radius of the blur

EdgeDetect: Detects edges in a blurred image.
input: `image_color` or `image_grey`
output: `image_grey`
parameters:
thr: minimum threshold with which to generate a responce.

SubMask: Subtracts the *image mask* from the passed image.
input: `<any image>`
output: `<corresponing image type>`

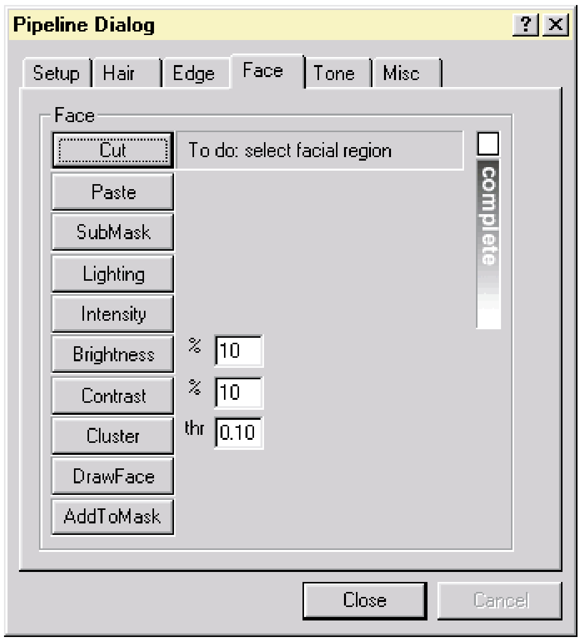
Despeckle: Removes lines from an edge-detected image that are less than a specified length.
input: `image_grey`
output: `image_grey`
parameters:
thr: the minimum line length to keep.

Cut/Paste: User is required to use an external program to erase unwanted edges.
input: `image_color` or `image_grey`
output: `image_color`

Intensity: Converts a color image into a grey-scale image.
input: `image_color`
output: `image_grey`

DrawEdges: Sends data to the output container.
input: `image_grey`
output: `<none>`

Figure 3.13: *The Edge dialog box.*



To do: select facial region

Brightness % 10

Contrast % 10

Cluster thr 0.10

Description of Operations:

Cut/Paste: User is required to use an external program to select the facial region.
input: `image_color` or `image_grey`
output: `image_color`

SubMask: Subtracts the *image mask* from the passed image.
input: `<any image>`
output: `<corresponing image type>`

Lighting: Normalizes the lighting on the image to ease later operations. User is required to select (left to right) points of various skin tone intensities.
input: `image_color`
output: `image_color`

Intensity: Converts a color image into a grey-scale image.
input: `image_color`
output: `image_grey`

Brightness: Changes the brightness of the image.
input: `image_grey`
output: `image_grey`
parameters:
%: precent change in brightness

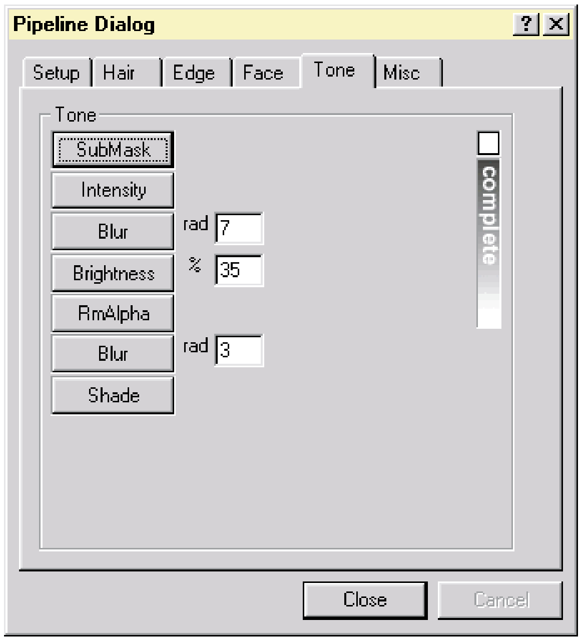
Contrast: Changes the contrast of the image.
input: `image_grey`
output: `image_grey`
parameters:
%: precent change in contrast

Cluster: Finds a cluster of pixels associated with each facial feature.
input: `image_grey`
output: `image_color_cluster`
parameters:
thr: grey-level above with which pixels can be considered for clustering.

DrawEdges: Sends data to the *output container*.
input: `image_color_cluster`
output: `<none>`

AddToMask: Adds the clustered eye regions to the *image mask*.
input: `<any image>`
output: `<none>`

Figure 3.14: *The Face dialog box.*



rad 7

% 35

rad 3

Description of Operations:

SubMask: Subtracts the image mask from the passed image.
input: `<any image>`
output: `<corresponing image type>`

Intensity: Converts a color image into a grey-scale image.
input: `image_color`
output: `image_grey`

Blur: Blurs the image.
input: `image_color` or `image_grey`
output: `image_color` or `image_grey`
parameters:
rad: radius of the blur

Brightness: Changes the brightness of the image.
input: `image_grey`
output: `image_grey`
parameters:
%: precent change in brightness

RmAlpha: Removes the masked area created by the **SubMask** step.

Blur: Blurs the image.
input: `image_color` or `image_grey`
output: `image_color` or `image_grey`
parameters:
rad: radius of the blur

Shade: Sends data to the *output container*.
input: `image_grey`
output: `<none>`

Figure 3.15: *The Tone dialog box.*

Chapter 4

The Hough Orientation Filter

The Hough orientation filter extracts local orientation from an image. More specifically, it computes the general orientation of the lines that lie in the small region surrounding each pixel in the source bitmap[1]. The result of the Hough orientation filter is a 2-dimensional array of 2-dimensional vectors. Each vector's direction and magnitude represent orientation and response strength respectively. Response strength can be defined as how strongly the lines in the region surrounding s pixel are oriented in a particular direction. An example showing the result of the Hough orientation filter is shown in Figure 4.1. Figure 4.1b shows the vector field after the Hough orientation filter is applied to the source bitmap shown in Figure 4.1a.

In short, the Hough orientation filter works by applying a small Hough transform to the region surrounding each pixel in an image. The result of the Hough transform is an analytic description of the lines contained in the region. The orientation of these lines can then be averaged to generate a vector that represents

[1]Note that the definition of orientation differs slightly from that of direction. Orientation is defined between 0 and π while direction is defined between 0 to 2π. For example, a vector pointed at 0 and a vector pointed at π have different directions but the same orientation.

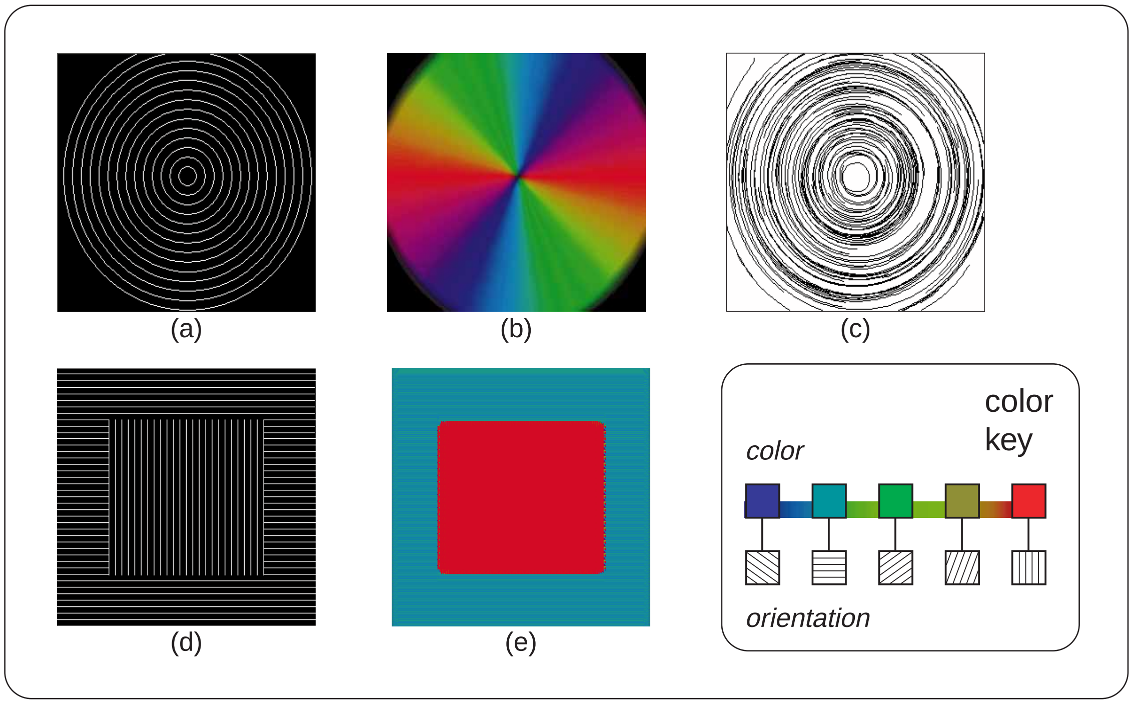


Figure 4.1: *A few sample images illustrating some applications of an orientation filter. (a) A source black-and-white bitmap. (b) The bitmap after being processed by the orientation filter. (c) A sample of using the orientation field to draw lines and create a sketch-like drawing. (d) A sample box-in-a-box image that is difficult for machines to interpret. (e) A image showing how an orientation filter can help remedy this machine vision problem.*

the local orientation.

One application of the Hough orientation filter is for artistic rendering. Figure 4.1c shows how streamlines can be used to trace an orientation field to generate a sketchy image. Another application for orientation filters, illustrated in Figures 4.1d,e, is to assist computers in detecting image features that are simple for the human visual system to see but difficult for machines to understand.

This chapter will begin by presenting a brief overview of the Hough transform in Section 4.2. Section 4.3 will detail how to extract general orientation information from the results of the Hough transform. Combined, these two concepts form the basis of the Hough orientation filter (Section 4.4). Finally, a few optimizations to the Hough orientation filter will be presented in Section 4.5. Potential areas for future optimizations will be discussed in Chapter 8.

4.1 Other Orientation Filters

As mentioned in Chapter 2, many other methods of computing local orientation have been proposed. The most general class of these methods are techniques based on the idea of applying a small set of filters at pre-specified angles and scales. The following section will review two techniques for computing local orientation in order to compare more common approaches to our novel technique. The first of these methods is application of the gradient operator. The second is the Quadrature Filter Set Method [28], which falls into the general class of techniques mentioned above.

The application of a standard gradient operator is probably the most straight forward method to compute local orientation of an image. The motivation behind this method is illustrated in the following example. Assume that we are trying to find the local orientation of the white lines in the image shown in Figure 4.1a. First, we treat the image as a height-field and apply a gradient operator throughout. This operation yields a set of vectors that point in an uphill direction and are perpendicular to the white lines in the source image. These vectors represent the exact local orientation that we are trying to compute. Thus, computing local orientation with the gradient operator simply involves applying the gradient op-

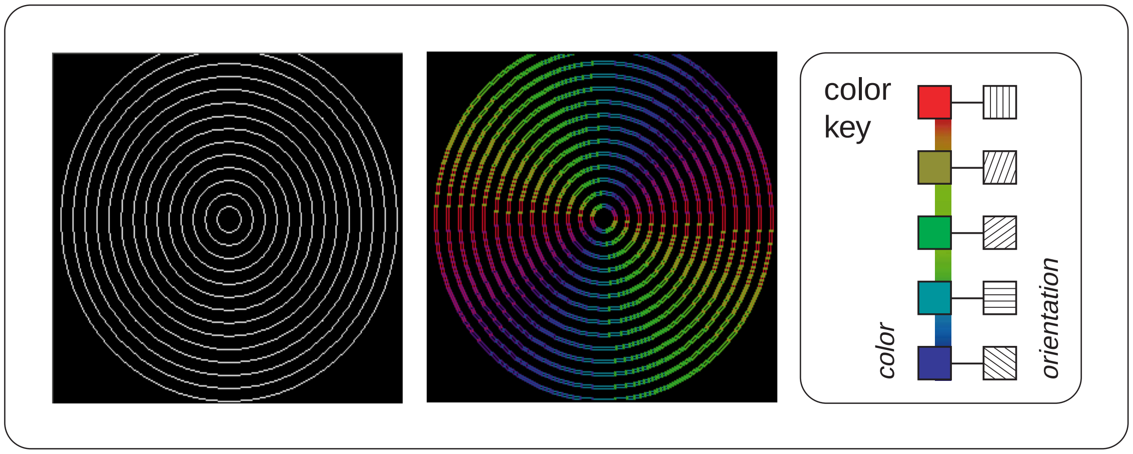


Figure 4.2: *The application of a gradient operator to compute orientation. The large vacancies are left in the orientation vector field (compared to Figure 4.1b) are due to the fact that the gradient operator is very localized.*

erator to each point in the image and then finding vectors that are perpendicular to those computed from the gradient (Figure 4.2). Although fast, the gradient operator only considers its eight surrounding pixels and is thus, too local of a feature [25]. For example, a purely random pattern with no specific orientation will generate a well defined gradient at each point in the image. Futhermore, large vacancies can also result in the final vector field due to this locality.

The Quadrature Filter Set Method [28] characterizes the most common method for computing local orientation. This method applies a set of filters to the small sub-region surrounding each pixel in a source image. Four directional filters with 45° increments in the directions of 22.5°, 67.5°, 112.5°, and 157.5° measure the orientational strength of the underlying region in its respective direction. The results of these filters are then averaged to find the general orientation of the sub-

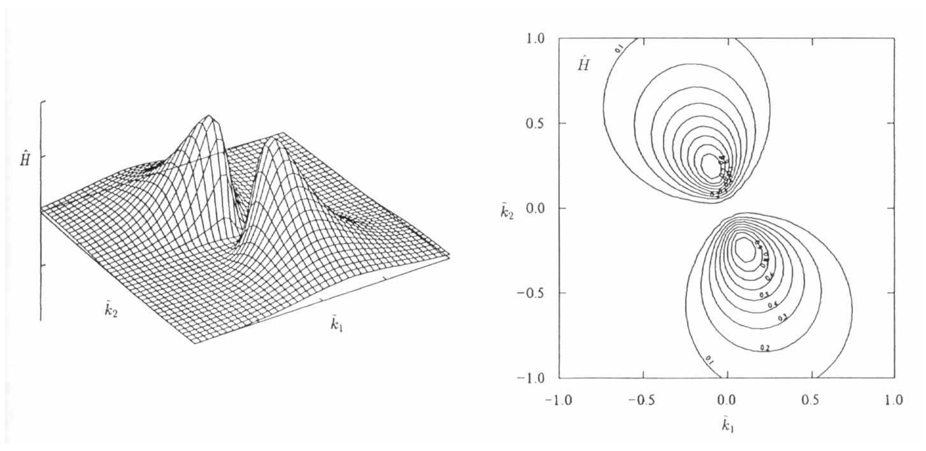


Figure 4.3: *A sample plot of the directional quadrature filter in the 112.5° direction.*

region. The filter used in this technique is known as a *quadrature filter*[2] and is illustrated in Figure 4.3.

Although many of the previously published methods for computing local orientation are applicable to our ultimate goal of artistically rendering hair, we chose to use the Hough transform to compute local orientation because of its robustness in finding lines in noisy images.

4.2 The Hough Transform

The Hough transform [21] is a standard method in machine vision to find a mathematical description for lines and shapes in an image. Although many variations to the Hough transform have been proposed (Section 2.2.1), the Hough orientation

[2] "A pair of filters is said to be in quadrature if they have the same frequency response but differ in phase by 90°. Such pairs allow for analyzing spectral strength independent of phase and allow for synthesizing filters of a given frequency response with arbitrary phase." [15]

filter employs a basic form of the Hough transform that finds lines in a black-and-white, 1-bit image. The following section will describe this basic form of the Hough transform and provide the motivation for using the Hough transform to find lines in an image in favor of a more brute-force method.

Assume that there exists an image that contains n points, and that we want to find the lines in the image that these pixels represent. One possible solution to this problem is to find all the lines determined by each pair of points in the image. This process generates $n(n-1) \sim n^2$ lines. The number of pixels that lie on each line must then be computed in order to find the strength or relevance of the line in the original image. This operation requires *the number of pixels* $\times$ *the number of lines* or $(n)(n(n-1)) \sim n^3$ total operations [16]. The Hough transform provides a much more computationally attractive solution.

Assume that we have a point, (x_0, y_0), which can be described by the equation, $y_0 = mx_0 + b$. An infinite number of lines exist through that point, and thus, there exists an infinite number of possibilities for m and b. Figure 4.4a illustrates a point (x_0, y_0) with 3 different lines passing through it. Now consider the same line equation rewritten as $b = -x_0 m + y_0$. Solving for m and b with a constant x_0 and y_0, we obtain, as before, an infinite number of (m, b) points. In the mb plane (also referred to as *parameter space*), plotting all possible m and b values creates a line with a slope, $-x_0$, and b-intercept, y_0. Figure 4.4b illustrates how the three lines from Figure 4.4a transform into a series of points that lie on a common line. This transformation, from lines in the xy plane that share a common point to points in the mb plane that share a common line, is the fundamental concept underlying the Hough transform.

Now, assume that we have two points in the xy plane, (x_0, y_0) and (x_1, y_1). Because these two points lie on a common line (Figure 4.5a), there exists an m_0 and b_0 such that $y_0 = m_0x_0 + b_0$ and $y_1 = m_0x_1 + b_0$ are true. Rewriting these two equations and generalizing for any m and b, we get $b = -x_0m + y_0$ and $b = -x_1m + y_1$. By plotting these two lines in parameter space, we find that they intersect precisely at (m_0, b_0) (Figure 4.5b). This observation is the basis of the Hough transform.

Assume, now, that we have n points along the line shown in Figure 4.5a, and that the equation of this line is unknown. Transforming these points into parameter space would create n lines that would all intersect exactly at the point, (m_0, b_0) (Figure 4.5b). Once the intersection point in parameter space is found, the original line equation can be analytically described by the standard line equation, $y = m_0x + b_0$.

To compute the Hough transform, we quantize the parameter space and divide it into a 2-dimensional array of accumulator cells. As previously discussed, every point in the source image transforms into a line in parameter space. However, now each line in parameter space is rasterized and drawn in such a way that instead of simply marking the location of individual pixels, accumulator cell values are incremented by one each time a line is drawn over it. Figure 4.5c illustrates a sample source image. Figure 4.5d shows two of the points from the source image converted into parameter space. Although not illustrated, the cell in which the two rasterized lines overlap has a higher count than its surrounding cells.

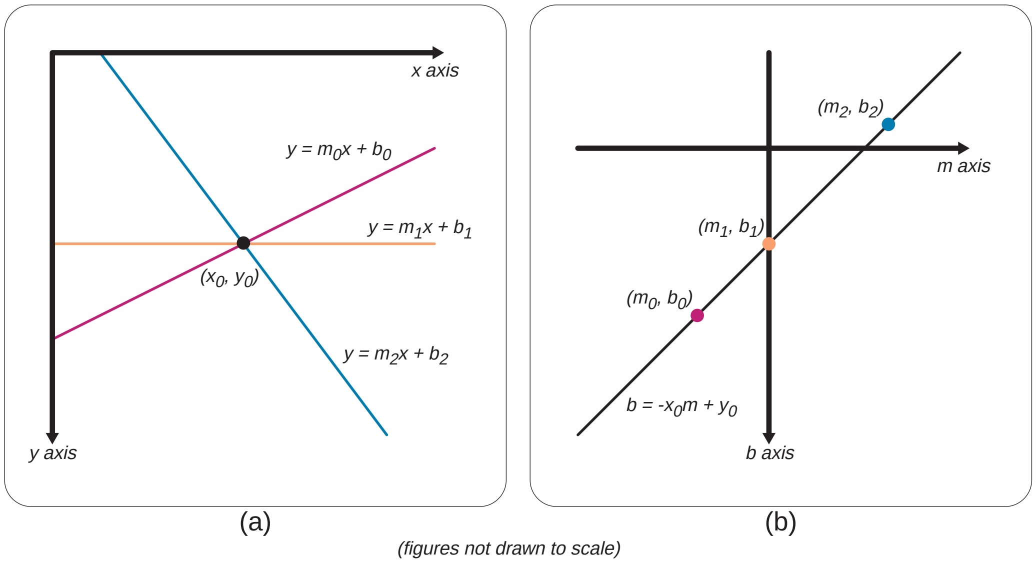


Figure 4.4: *Illustration of the Hough transform. (a) A set of lines that pass through the point* (x_0, y_0)*. (b) A figure illustrating how each of these lines can be represented as a point along a common line in parameter space.*

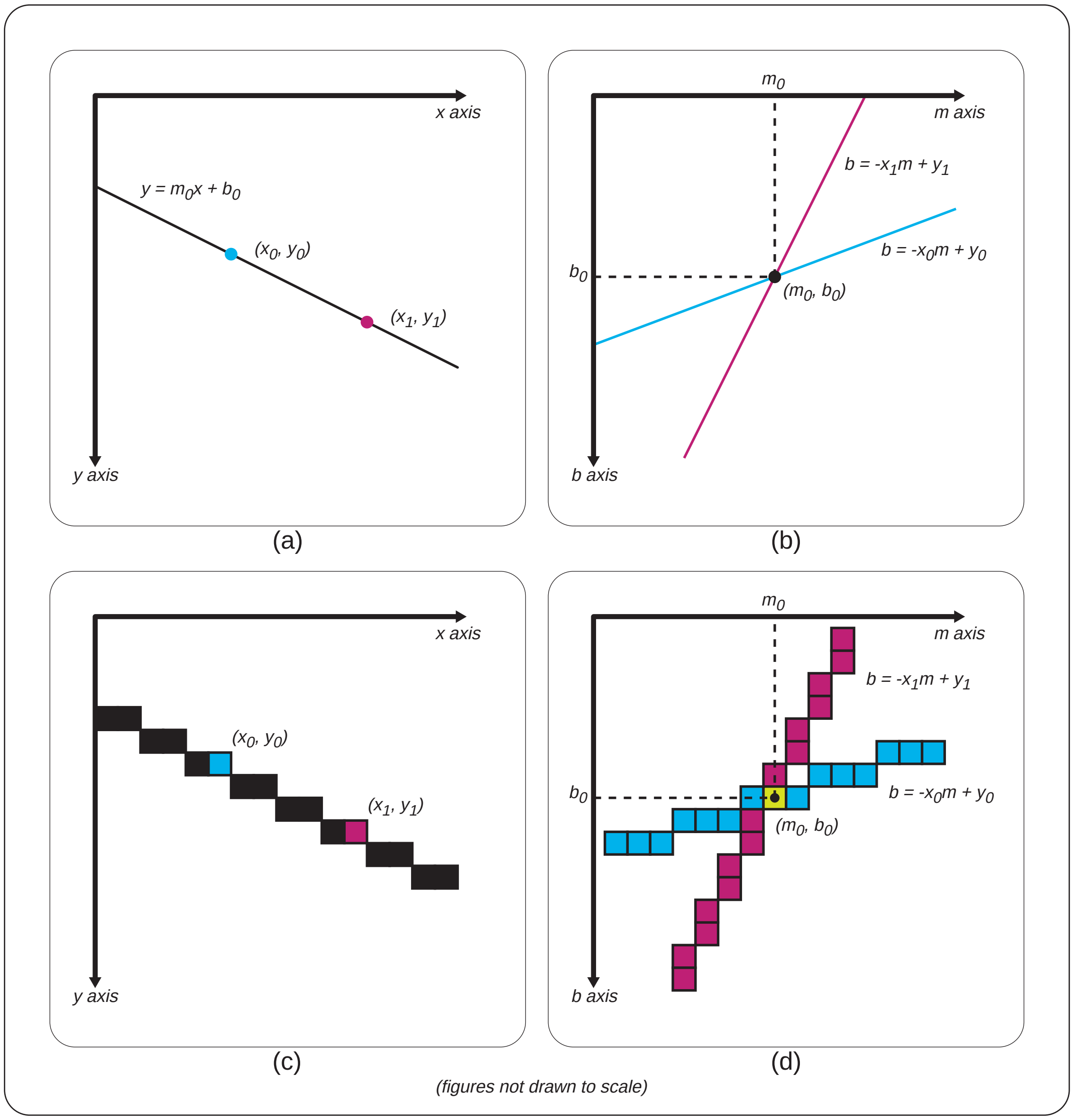


Figure 4.5: *Illustration of the Hough transform. (a) A line containing two points,* (x_0, y_0) *and* (x_1, y_1)*. (b) The two points converted into lines when transformed into parameter space. (c) The same two points, except now they lie in a discrete, rasterized line. (d) The two points converted into parameter space. The center bin represents an accumulator cell which has been incremented twice because it is overlapped by two lines. The cell with the highest count represents the slope and intercept of the original line.*

Once all the pixels in the source image have been drawn in parameter space, we can extract line information by looking for the accumulator cells with large values. A significant line in the source image will produce many lines in parameter space that repeatedly overlap at a single point. From this point, the slope and intercept (m and b) of the original line in the source image can be extracted.

A problem that arises when using the slope-intercept form of the line equation, $y = mx + b$, is that both the slope, m, and the intercept, b, approach infinity as the line becomes vertical. This is a problem for the Hough transform since we can not have an infinitely large parameter space. Moreover, even using a relatively large parameter space to capture near-vertical lines may be too computationally intensive and require too many resources for some tasks. Furthermore, scaling the parameter space to represent a large area with a small array undesirably reduces the resolution and accuracy of identifying non-near-vertical lines. To avoid these problems, we use the normal representation of a line, $\rho = x \cos\theta + y \sin\theta$ (Figure 4.6a) [13].

With the normal representation of a line, the Hough transform is computed as before, except that now, each (x, y) is transformed into a sinusoidal curve in the $\rho\theta$ plane (Figure 4.6b,c). As before, the application of the Hough transform requires every point in the source image to be transformed into a curve in parameter space. Values are accumulated and cells with large values represent significant lines in the source image. Also note that θ is limited to range from $-\theta/2$ to $\theta/2$. Bounding the range of θ prevents the computation of redundant information. For example, a line defined by $\rho = x \cos\theta + y \sin\theta$ is the same as the line defined by

$-\rho = x\cos(\theta + \pi) + y\sin(\theta + \pi)$ and $\rho = x\cos(\theta + 2/\pi) + y\sin(\theta + 2\pi)$.

Figure 4.7a shows a sample source image and Figure 4.7b shows the same image after the application of the Hough transform.

4.3 Orientation Extraction

Once the Hough transform has been computed, the orientation information can be extracted. The generation of the parameter space in the $\rho\theta$ plane simplifies this process because accumulator cells can be directly indexed by θ and no additional computations are required to extract angular information. A further simplification comes from the fact that we are only concerned with orientation. This focus on orientation allows us to disregard the ρ axis. Thus, the first step in orientation extraction is to "squash" the ρ dimension of the $\rho\theta$ parameter space to produce a 1-dimensional array of orientation response strengths indexed by θ. This reduction is done by summing, for each θ, the corresponding ρ accumulator cell values that rank among the largest 5 percent in the entire parameter space[3]. The computed sum is then stored in the orientation response strength array. Each value in the array represents the strength of the orientation response for that particular θ. For a source image containing a set of lines of roughly equivalent orientations, most of the values in the resulting response strength array will be zero.

For each non-zero response, we generate a vector whose orientation is based on the corresponding θ index and whose magnitude is based on the array bin

[3]5 percent is an arbitrary value which produced good results during experimentation.

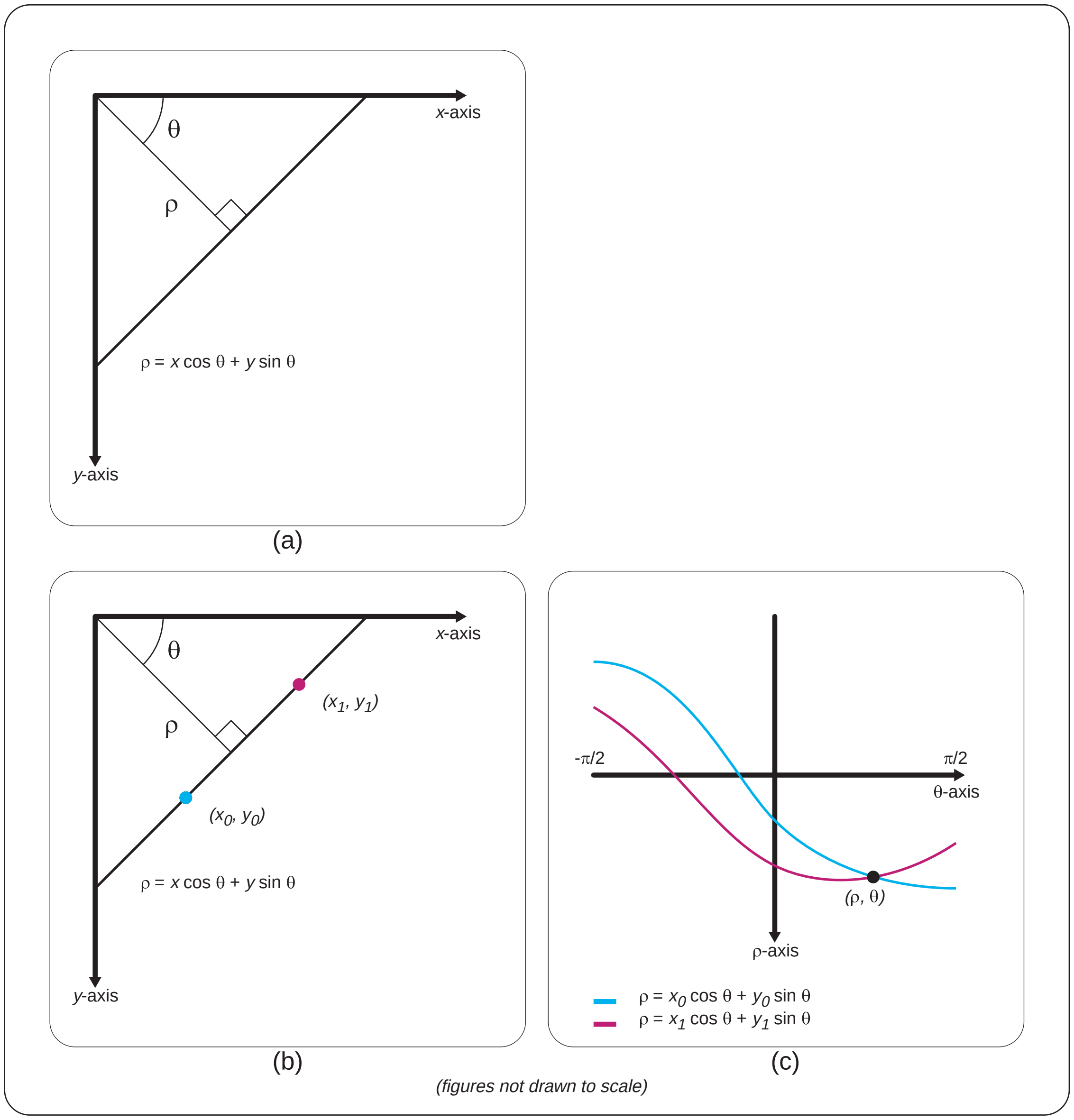


Figure 4.6: *Illustration of how the normal representation of a line is used with the Hough transform. (a) An illustration of the normal representation of a line. (b) Two points* (x_0, y_0) *and* (x_1, y_1) *that lie on a line,* $\rho = x\cos\theta + y\sin\theta$*, where* ρ *and* θ *are constant. (c) Two points,* (x_0, y_0) *and* (x_1, y_1)*, transformed into the parameter space curves,* $\rho = x_0\cos\theta + y_0\sin\theta$ *and* $\rho = x_1\cos\theta + y_1\sin\theta$*. Note that the two curves intersect at the point* (ρ, θ)*.*

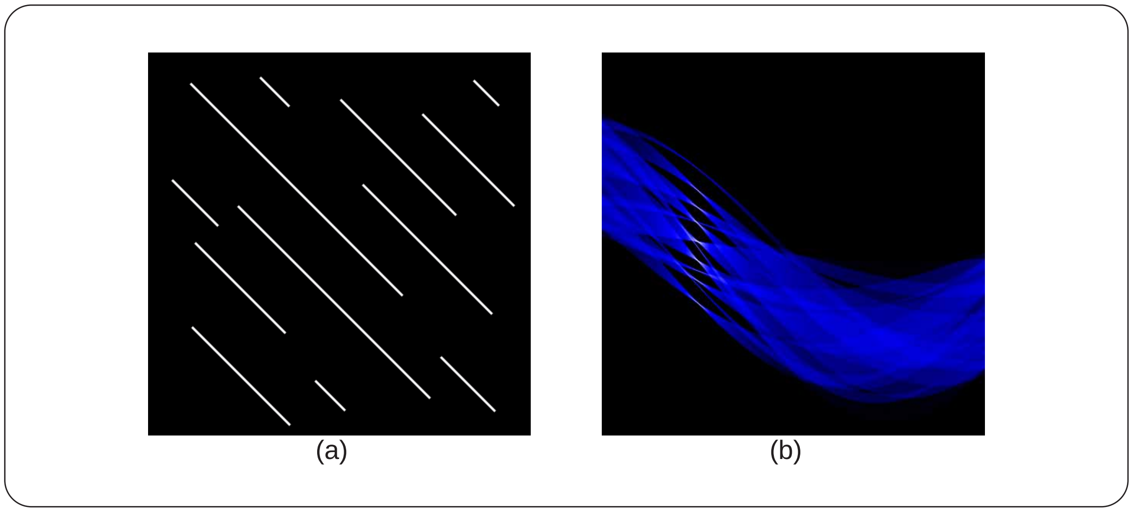


Figure 4.7: *A sample result of the Hough transform applied to an image containing lines. (a) The original image. (b) The resulting accumulator cells after the application of the Hough transform. The lighter colored pixels represent accumulator cells with greater bin counts. Note that because the lines in the source image are oriented in the same direction, the peak points in the parameter space share the same θ.*

value. The resulting set of vectors must then be averaged to find the average orientation of the source image. Unfortunately, we cannot simply add the vectors and divide by their total magnitude to find the average orientation. Figure 4.8a-c illustrates this dilemma. In Figure 4.8a, we have vectors representing two different orientations. In Figure 4.8b, we average the two vectors, and in Figure 4.8c, the resulting orientation is shown. However, this orientation is incorrect. The correct solution is shown in Figure 4.8g. Jähne [25] describes a technique for properly working with orientation.

Because the range of orientation is limited to π radians rather than 2π radians, it is helpful to multiply the orientation of a vector by 2 before applying operations to it. This adjustment in the vector's angle introduces the cyclic behavior necessary to reason with orientation using standard geometric operations (i.e. two vectors directed at 0 and π radians should be mathematically treated as the same orientation in the same way that two vectors directed at 0 and 2π are treated as the same direction.). Figures 4.8d-g illustrate how this technique is used to average orientation vectors. Before averaging, each vector's direction with respect to the $0°$-axis is first doubled (Figure 4.8d). The new vectors are averaged (Figure 4.8e), and the resulting vector's direction is then halved (Figure 4.8f). Figure 4.8g shows the computed average orientation.

4.4 The Hough Orientation Filter

Once we are able to extract orientation information from the parameter space of the Hough transform, the application of the Hough orientation filter is relatively straight forward. As previously described, for every pixel in an image, a Hough transform is applied to the surrounding region and the orientation is extracted. By applying this process everywhere, the local orientation at each pixel is found.

There are two main parameters of this process. The first is the size of the surrounding region around each pixel to which the Hough transform will be applied, and the second is the resolution of the Hough transform's parameters space. In general, the size of the Hough transform window should be wider than the average distance between the lines in the source image. A smaller window size produces black spots in the resulting orientation vector field where the Hough transform fails to detect any significant lines and no orientation can be extracted. Furthermore, a small Hough window also creates quantization errors that reduce the accuracy

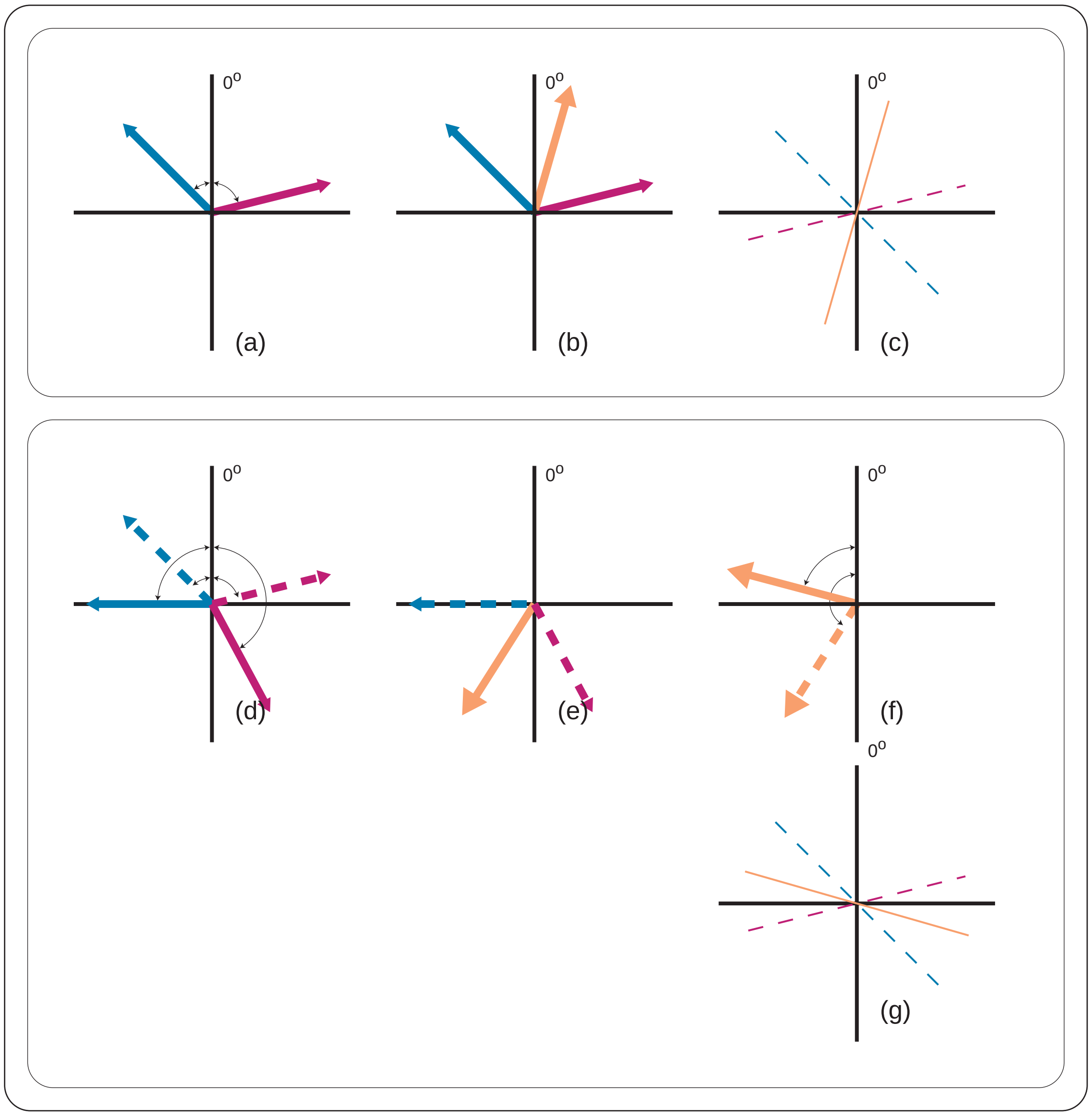


Figure 4.8: *Illustration of how to average orientation vectors. Figures (a)-(c) show an example where simply averaging vectors produces an incorrect average orientation. Figures (d)-(f) show how this problem can be solved by doubling vector angles before the averaging process and then halving them afterwards.*

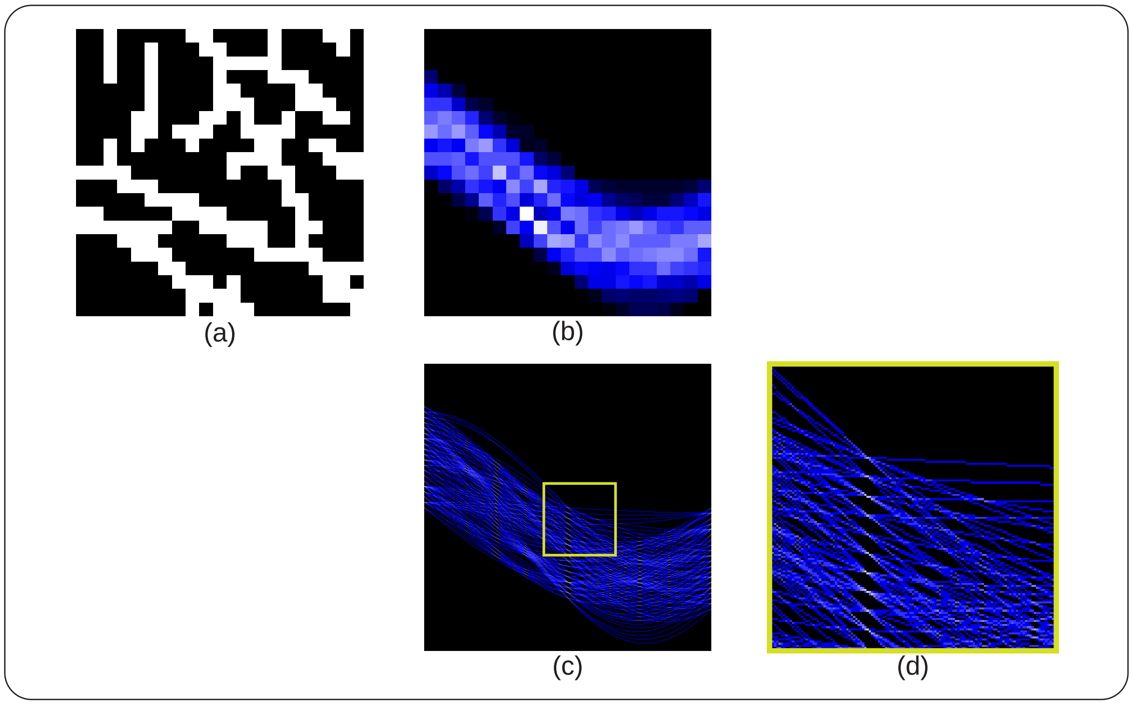


Figure 4.9: *A set of images illustrating how quantization errors occur when applying the Hough transform to a small image with a relatively large parameter space. (a) The 21×21 source image. (b) The results of the Hough transform applied to a 21×21 accumulation buffer. Note that the general trends in the source image are captured well. (c) The Hough transform with a relatively over-sized accumulation buffer. (d) A magnified region of the parameter space shown in (c). Note how the quantization errors cause the curves drawn in parameter space to cross at incorrect points.*

of the final result.

The second parameter, the resolution of the Hough parameter space, should be chosen to be roughly the same size as the original image. Too large of a parameter space introduces quantization problems that lead to inaccurate results and increased computational time (Figure 4.9). A parameter that is too small reduces the accuracy and resolution of the Hough transform, although having the desired effect of reducing computational time.

When the Hough orientation filter is applied to lines detected in a natural image, the resulting vector field is generally very noisy. Before the vector field can be used for our application of artistically rendering hair, the vector field must first be blurred.

To smooth the vector field, we augment a standard Gaussian blur with two operations. The first operation doubles a vector's angle with respect to the 0°-axis for every vector in the vector field. This operation guarantees the proper averaging of vectors as previously described. Next, each vector's magnitude is squared to more heavily weight vectors that represent strong orientation responses. The resulting vector field is then smoothed with a relatively large Gaussian kernel. Experimental results have shown that a standard deviation of 15 produces results that adequately smooth the vector field while perserving the major details of the source image. Once smoothed, the vectors are returned to their original form.

Figure 4.10b shows the result of applying the Hough orientation filter to the image in Figure 4.10a. Figure 4.10c shows the result of smoothing the vector field resulting from the Hough orientation field. Figure 4.10d shows how the smoothed vector field can be used to draw hair.

4.5 Optimizations

Computing a Hough transform for each pixel in a standard image composed of a few thousand pixels is a very computationally intensive task. Several strategies have been implemented to reduce this comutational process. Look-up tables are used to avoid the repeated computation of sines and cosines when computing curves in the

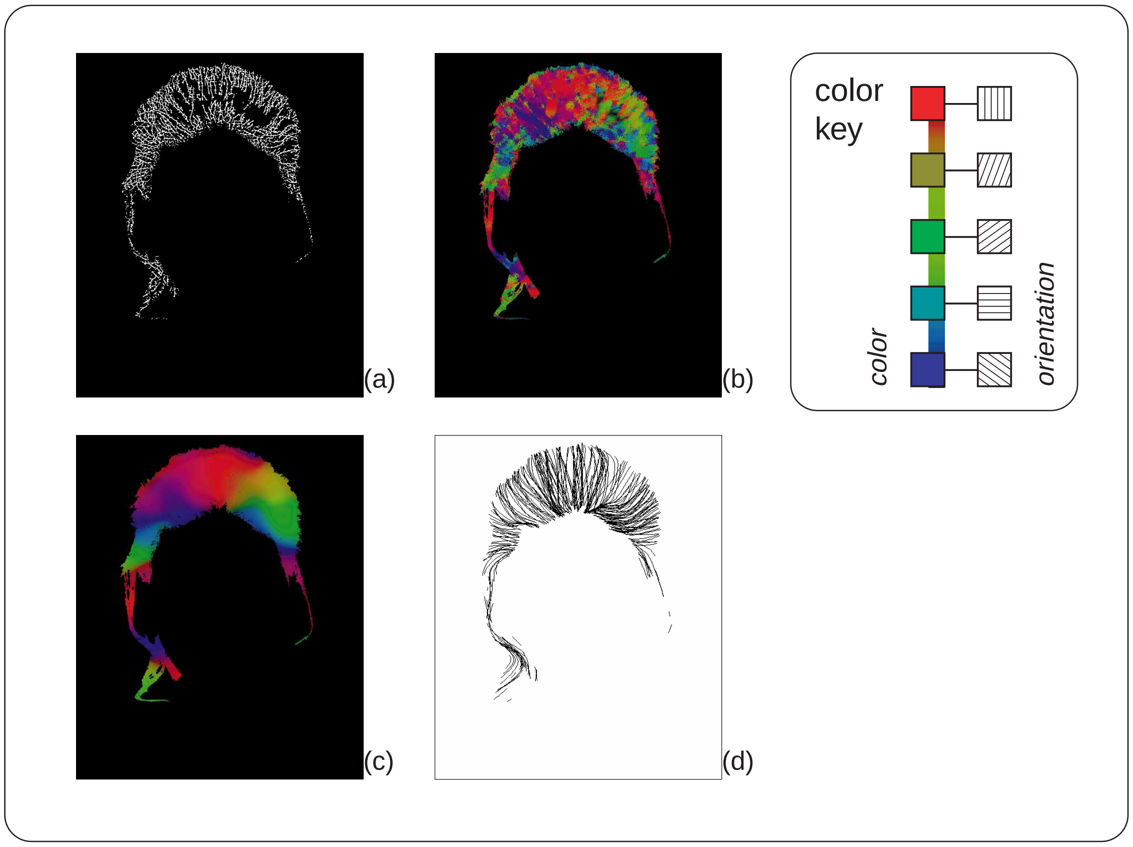


Figure 4.10: *A sample result of the Hough orientation filter. (a) The source image. (b) The result of applying the orientation filter to the source image. (c) The orientation field after smoothing. (d) A sample of using the orientation field to draw sketchy, hair-like lines.*

$\rho\theta$ parameter space and provide a signifcant performance increase. Furthermore we parallelize the Hough orientation algorithm to take advantage of machines with multiple processors. This parallelization is done by simply dividing scan-lines across different threads. These threads are then distributed between processors by the operating system.

Chapter 5

Facial Feature Extraction

The extraction of facial features is one of the most important steps in the artistic rendering of portrait photographs. Specifically, we wish to isolate the six major features of the face: the eyebrows, eyes, nose, and mouth. Each of these regions must be isolated so that each feature can be treated independently and rendered with a specialized drawing algorithm. Details of the actual shading algorithms can be found in Chapter 6.

As described in Chapter 2, our facial feature extraction process is based on the assumption that facial features are generally darker than the skin tones in the surrounding face. Although this assumption may not hold true for all faces, it is generally a valid assumption and commonly used in many vision systems.

The following chapter will describe the process of finding facial features. Section 5.1 will describe a simple, user-interactive method to robustly normalize the lighting in a portrait photograph. This normalization process ultimately achieves more accurate results. Section 5.2 will then discuss the algorithm we employ to isolate the facial features.

5.1 Lighting Normalization

The facial feature extraction process developed for this thesis is based on a classic machine vision technique known as thresholding. Because we assume that facial features are darker than their surrounding regions, we can threshold the facial region to find all the pixels that lie above and below a specified grey-level. Once the image is thresholded, the darker regions can be isolated and assumed to compose the facial features. Individual features are then found by first grouping dark regions into clusters and comparing the relative positions of the clusters to one another.

One of the basic premises of our artistic rendering system is that we are working with portrait photographs. This premise eliminates the problem of needing to work with photographs taken under bad lighting conditions. However, even with the highly controlled lighting environment in the portrait studio, there are still shadowed regions in the portrait that must be corrected before the feature extraction process can be applied. Figure 5.1 shows an example of a typical portrait photograph. Although seemingly uniform, notice how the colors across the face vary greatly from the highlight on the forehead to the darker regions on the sides.

The lighting across the face must be normalized before the thresholding process can be applied because of this inconsistency. In general, however, this normalizing of lighting is a non-trival task. To help resolve this problem, we make the simplifing assumptions that the head is roughly cylindrical and that the most significant lighting for the portrait is positioned roughly in the horizontal plane around the head. Based on these assumptions, we define a simple, user-interactive process

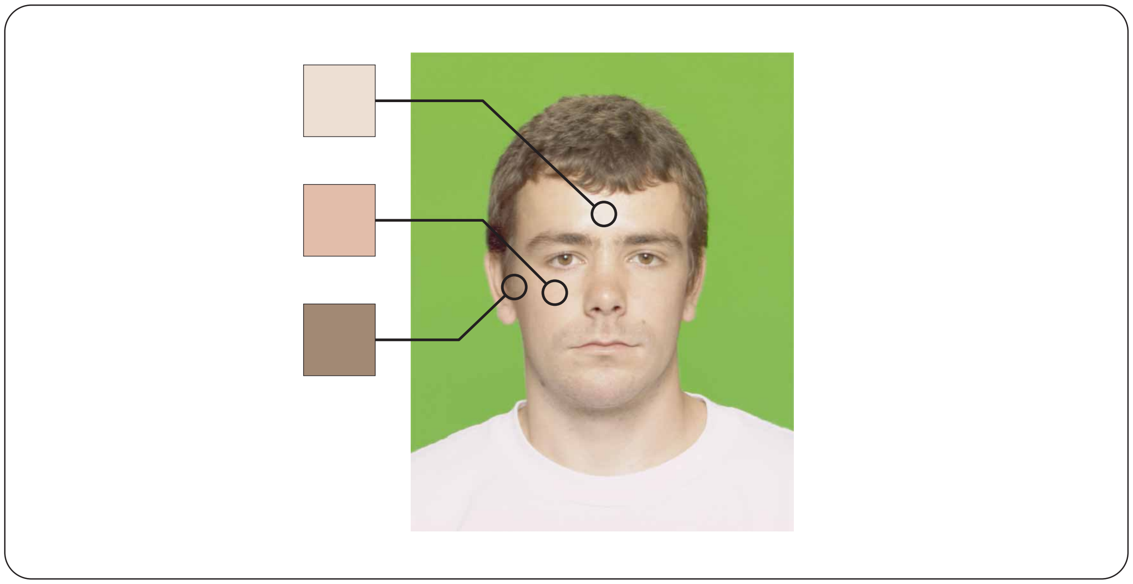


Figure 5.1: *A figure showing how much color changes across the face of a portrait photograph. Note that the skin color at the center of the forehead is much lighter than that at the edge.*

to normalize the image. The process requires the user to pick five points across the face in regions of different tonal values. Generally, choosing points from the following regions works well: the left edge of the left cheek, the center of the left cheek, the center of the forehead, the center of the right cheek, and the right edge of the right cheek. The red crosses in Figure 5.2a show some typical sample points.

Once the points have been picked, the colors of the points are interpolated to generate a color gradient image (Figure 5.2b). The interpolation process works by setting the color of each column in the gradient image to the linearly interpolated color of its two closest sample points. The columns that lie outside the group of sample points are simply set to the color of their closest point.

The gradient image is then subtracted from the source image (Figure 5.2c), and the result is then inverted to provide better visualization (Figure 5.2d). As a final step, the normalized image is converted to grey-scale (Figure 5.3a) and the brightness and contrast of the original image are increased to wash out any remaining dark regions (Figure 5.3b). By the end of this step, none of the facial features should be connected by dark regions (i.e. the facial features should form a disjoint set of dark areas in the image). Any dark regions that are not related to a specific facial feature will confuse the feature clustering algorithm (Section 5.2). Figured 5.3c shows the grey-scale version of the original image. Figure 5.3d shows the brightness and contrast of the original increased. Note how in Figure 5.3d there are still unwanted dark areas around the edges of the face and between the eyes and eyebrows.

5.2 Feature Clustering

Once the lighting in the facial image has been properly normalized, the thresholding and clustering algorithms can be applied. Thresholding the image simply involves removing all pixels that are above a specified grey-level. This process removes the majority of the pixels that represent skin and leaves behind islands of pixels that represent facial features.

Once the image is thresholded, the remaining pixels must be grouped into clusters that represent significant facial features. A recursive region growing algorithm is applied to do this clustering. The algorithm works by scanning through the image and searching for pixels that have not been removed by the thresholding process. When an active pixel is found, a cluster is grown by recursively adding

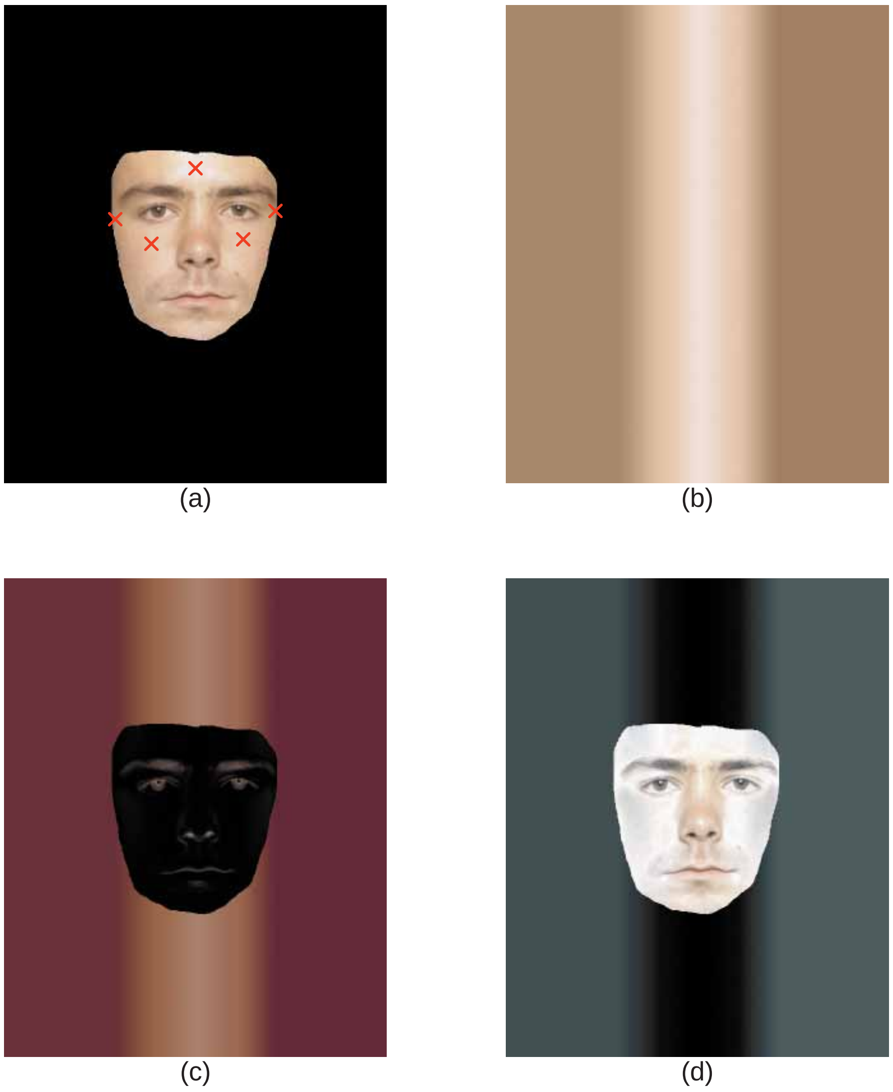


Figure 5.2: *A series of images illustrating the lighting normalization process. (a) The original image (after the selection process). The red crosses on the face show a set of typical sample points. (b) The color gradient generated from these sample points. (c) The result of subtracting (b) from (a). (d) The results of the subtraction process inverted.*

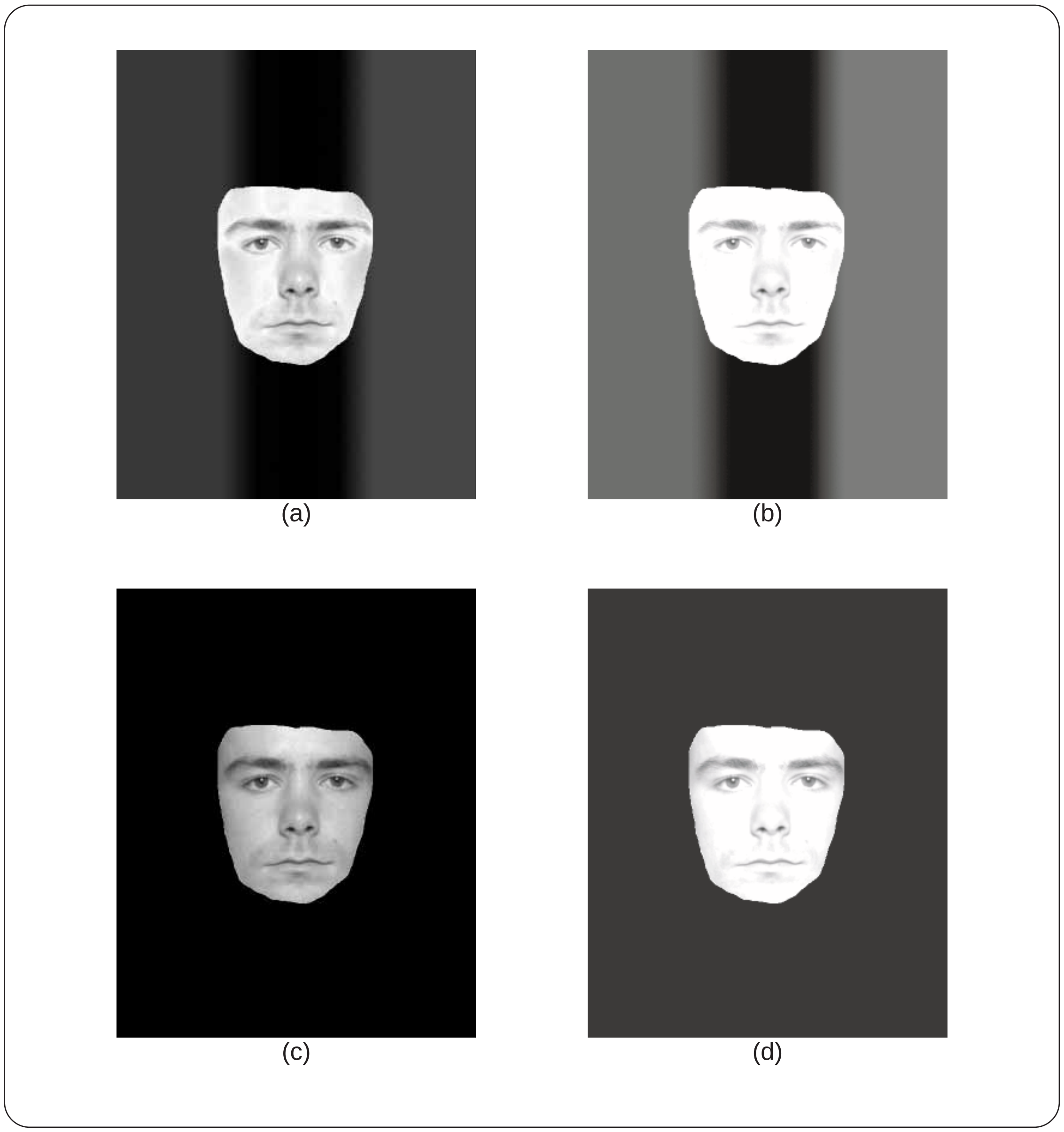


Figure 5.3: *A comparison of an image with and without the normalization process. (a) The grey-scale version of the result of the normalization process. (b) The brightness and contrast of the image increased. Note how the facial features now form disjoint dark regions in the image. (c) The grey-scale version of the original image without processing. (d) The brightness and contrast of the original image increased. Note how there are still dark areas that connect the eyes and eyebrows.*

neighboring pixels. Pixels that have been added to a cluster are marked "visited" so that they will not be referenced again. This process continues until the entire image has been clustered.

Six clusters, representing facial features, will ideally be isolated after the completion of this process. In general, however, many more clusters result, and further processing is required to obtain a clean representation. To solve this problem, we make use of the observation that in general, after the clustering proces, each facial feature decomposes into one large cluster surrounded by a few scattered pixels. Thus, we define a reclusterizing process that involves the following steps: 1) sorting the clusters based on their size, 2) keeping the six most significant clusters and breaking down the remaining clusters into individual pixels, and 3) merging these declustered pixels with the cloest remaining significant cluster.

Figure 5.4 shows the results of applying this process to the images in Figure 5.3b,d. Note that when the clustering process is applied to the non-normalized image, an incorrect clustering results.

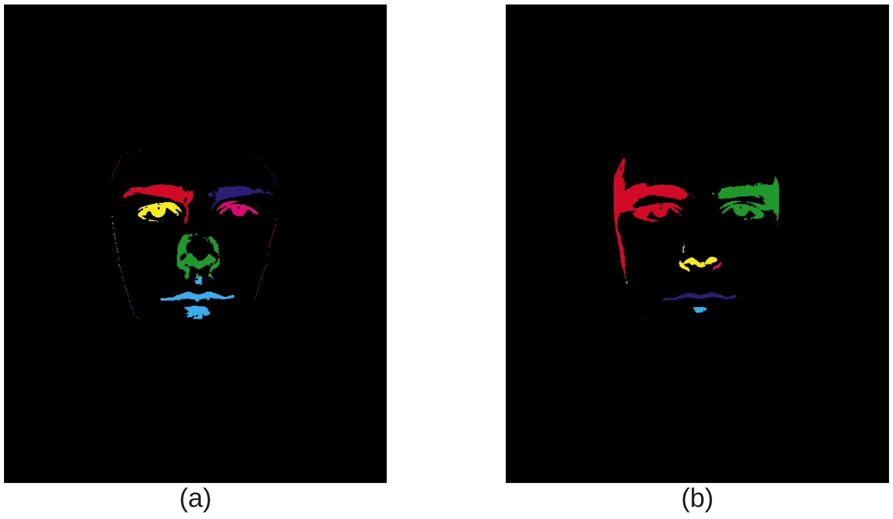


Figure 5.4: *Results of the facial feature clustering process. (a) The result of applying the clustering process to the image in Figure 5.3b. Different colored regions represent different clusters. (b) The result of applying the process to the image in Figure 5.3d. Note how the clustering algorithm fails when the lighting normalization process is not applied.*

Chapter 6

Image Drawing and Shading

The final step in the artistic rendering process is to generate a drawing from the data computed in the five major computational steps discussed previously (Chapter 3). The data from these steps include the following: 1) the region of the source image containing the background pixels, 2) an orientation field representing the hair region, 3) a black-and-white image representing the signifcant edges of the ears, chin, neck, and shoulders, 4) a set of clusters that identifies the facial features, and 5) a blurred grey-scale image that will be used to create the tonal shading of the face.

The following chapter will begin by presenting two supporting functions for the drawing and shading algorithms described in Section 6.3. The first supporting function, the image thresholding algorithm, is used by the facial feature drawing algorithm and will be described in Section 6.1. Section 6.2 will then present the second supporting function, a technique to simulate how a human naturally shades regions. This hand-shading technique is used by both the background shading and the facial feature drawing algorithms.

As a final note, our artistic rendering system focuses only on the actual drawing algorithms. The simulation of natural media is left to an external program. Our system generates scripts that contain information such as pen-placement and brush-selection commands. These scripts are then imported into MetaCreation's Painter [37] software, which interprets the commands to draw the final image. As mentioned in Chapter 2, Painter is a natural media painting program that provides a rich set of brushes including charcoal, pens, pencils, and paints.

6.1 Image Thresholding

Image thresholding involves finding a set of thresholds that mark significant spikes or features in a histogram of pixel values from a grey-scale image. Once computed, these thresholds can be used to group pixels that are of a similar intensity. In machine vision, thresholding is a common technique used to segment images. For example, if we had an image of a black box on a white background, the image could be thresholded at 50 percent grey to separate the box from the background. In this case, image thresholding is relatively simple. However, in natural images, image thresholding is a much more difficult problem.

Natural images are often noisy and seldom composed of a discrete set of solidly colored objects. The resulting histograms computed from these images are thus equally noisy, and finding thresholds is not simply a problem of searching for local minima. To compute thresholds, we use a simplified version of the color image segmentation algorithm introduced by Lim *et al.* [33].

Lim's segmentation algorithm, in general, works by creating a color histogram of the pixels in the source image and then searching for large clusters in the histogram. The centers of these clusters are then used to group pixels in the original image. One key component of Lim's algorithm is the use of *scale space filtering* [52] to threshold histograms computed from the individual red, green, and blue color channels of a source image. This is the thresholding scheme we use to segment our grey-scale images.

Once we have a thresholding algorithm, the shading of the eyes, nose, and mouth can be done by separating each facial feature into a disjoint set of regions. Each region is characterized by a particular grey value and can then be shaded with the appropiate brush and tone.

The following sub-section will give an overview of scale space filtering and how it can be used to threshold images. Further details of this process can be found in the references cited above.

6.1.1 Scale Space Filtering

Assume that we have a noisy histogram (Figure 6.1a) created from an image that we wish to threshold. To do this, we must find the boundaries of the significant features in the histogram. One solution to this problem is to smooth the histogram with a Gaussian kernel to remove unwanted noise. Once smoothed, the histogram can be thresholded by simply searching for local minima. Unfortunately, it is difficult to know what value to choose for the standard deviation of the Gaussian kernel *a priori*. Under-smoothing the histogram will leave unwanted noise and lead to the false identification of features. Over-smoothing will eliminate noise but remove desired features.

Witkin [52] proposed a method, referred to as *scale space filtering*, to help solve this problem. Assume again that we have a noisy histogram (Figure 6.1a). Now, rather than creating a single smoothed version of the histogram, we compute a series of smoothed histograms using successively larger standard deviations for the Gaussian kernel. This set of histograms, each represented at a different scale, forms the *scale-space image*. Figure 6.1 shows a set of smoothed histograms with their associated σ's. Figure 6.2a shows the same set of histograms grouped together to form a surface. Brighter colors represent higher regions in the histogram. The horizontal axis represents grey levels from 0 to 1. The vertical axis represents σ increasing from 0.2 to 5.2.

Next, the points of inflection of each histogram are found by computing the second derivative and searching for zero crossings. Figure 6.2b shows the points of inflection, from the set of smoothed histograms, connected to form lines. Notice how the points of infection are generally grouped as pairs that disappear as the standard deviation for the Gaussian kernal, σ is increased. This pairing comes from spikes and dips in the original historgram that flatten and disappear as σ is increased. Large features in the histogram that remain at the maximum scale create point-of-inflection pairs that do not fade away. Once the point-of-inflection lines have been computed, they are straightened to facilitate later computation (Figure 6.2c).

The final step in scale space filtering is to identify long regions in the straightened point-of-inflection data. These long regions are refered to as regions of *maxi-*

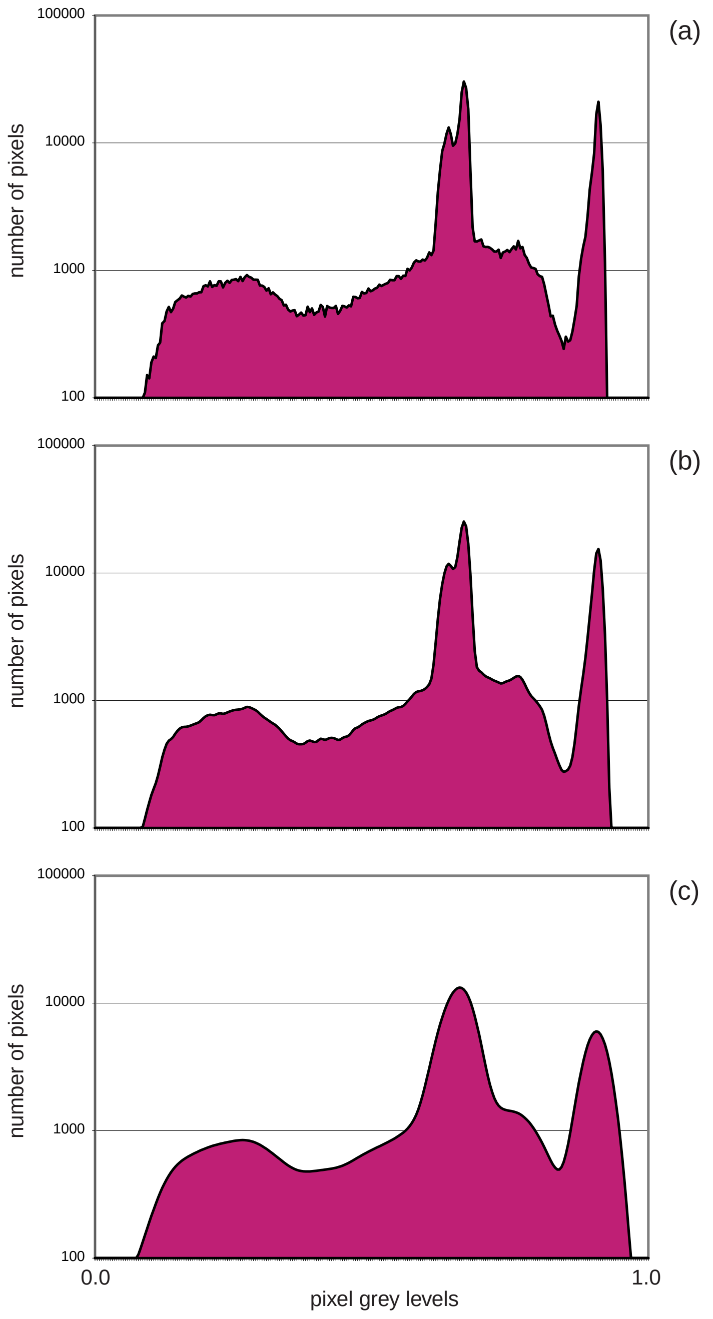


Figure 6.1: *A few histograms at varying scales. (a) The original histogram computed from a grey-scale portrait. (b), (c) The histogram smoothed with a Gaussian kernel of* $\sigma = 1.2$ *and* $\sigma = 5.2$ *respectively. Note how small features drop out at* σ *increases.*

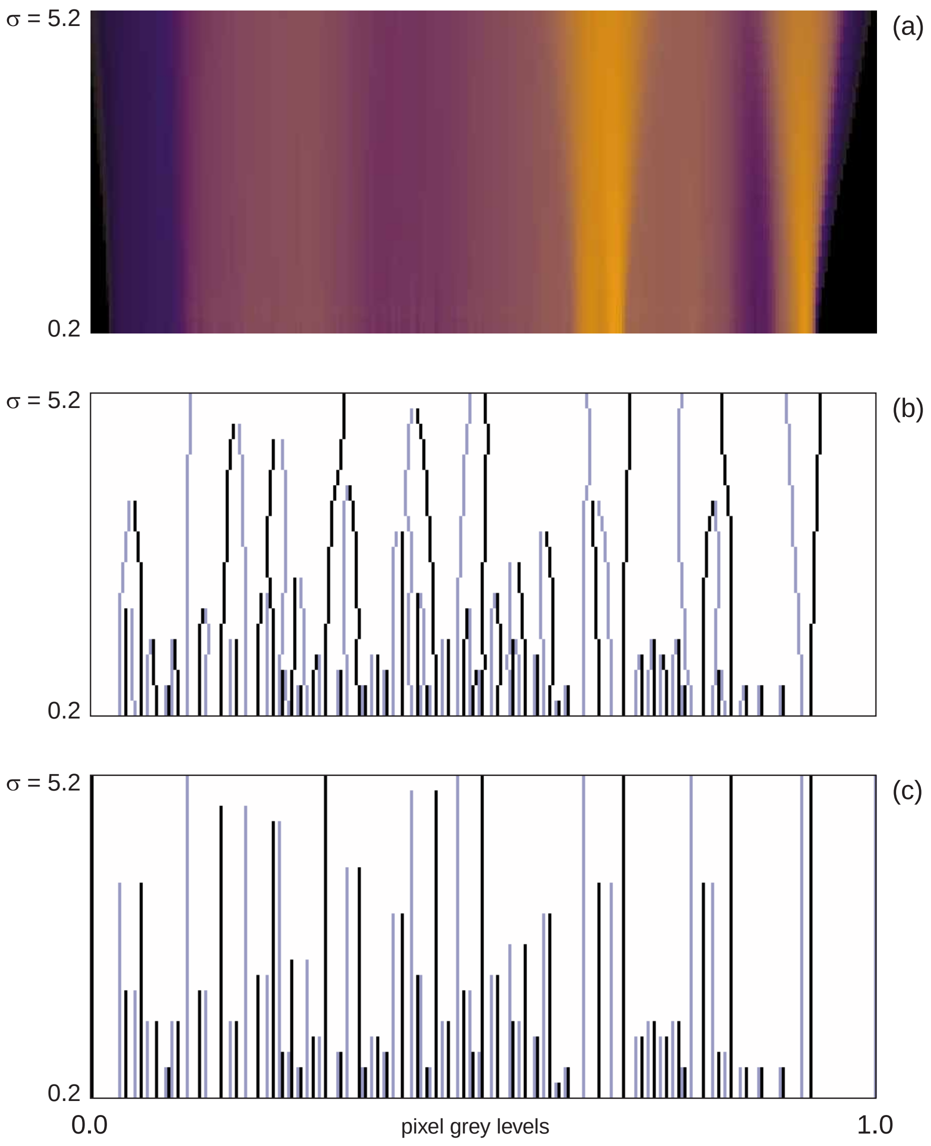


Figure 6.2: *Plots showing the scale-space image and point-of-inflection data. (a) The height field representing the scale-space image for the sample histogram in Figure 6.1a. (b) Lines representing the points of inflection of the scale-space image. (c) The straightened point-of-inflection lines. Light blue lines in the figure represent points of inflection in the surface to the left of local maxima. Black lines represent points of inflection to the right of local maxima.*

mum stability and represent features in the source histogram that are most resilient to the smoothing process. To identify maximum stablity regions in the straightened point-of-inflection data, we first construct an *interval tree.* In general, each interval parents three sub-intervals. Figure 6.3a illustrates one branch of the interval tree for our sample data. The white rectangle at the top of the unshaded region in the figure represents the root interval. From the top down, we search for nodes in the tree that are longer than the average length of their children. These nodes are shown in Figure 6.3b. Finally, because we are only concerned with data containing local maxima, we can quickly eliminate half of the intervals.

Once the maximum stability regions have been identified, the boundaries of these regions are used to threshold the image. The number of pixels that fall into each of these regions is counted. Those regions that contain a number of pixels exceeding a prespecified amount are used as a basis for the thresholding process. Pixels in the image are grouped to the basis grey-level that they lie cloest to. The final result is a tresholded image.

6.2 Hand Shading Simulation

In order to shade individual regions in our final drawing, we need a shading method that looks and feels natural. Filling an area with a simple horizontal or vertical pattern produces a very rigid and machine generated feel. However, by simulating the method in which a human hand shades a region, we can create shaded areas that appear very natural. Figure 6.4 shows a sample of our shading algorithm. Essentially, the shading algorithm works by drawing a set of arcs that are connected

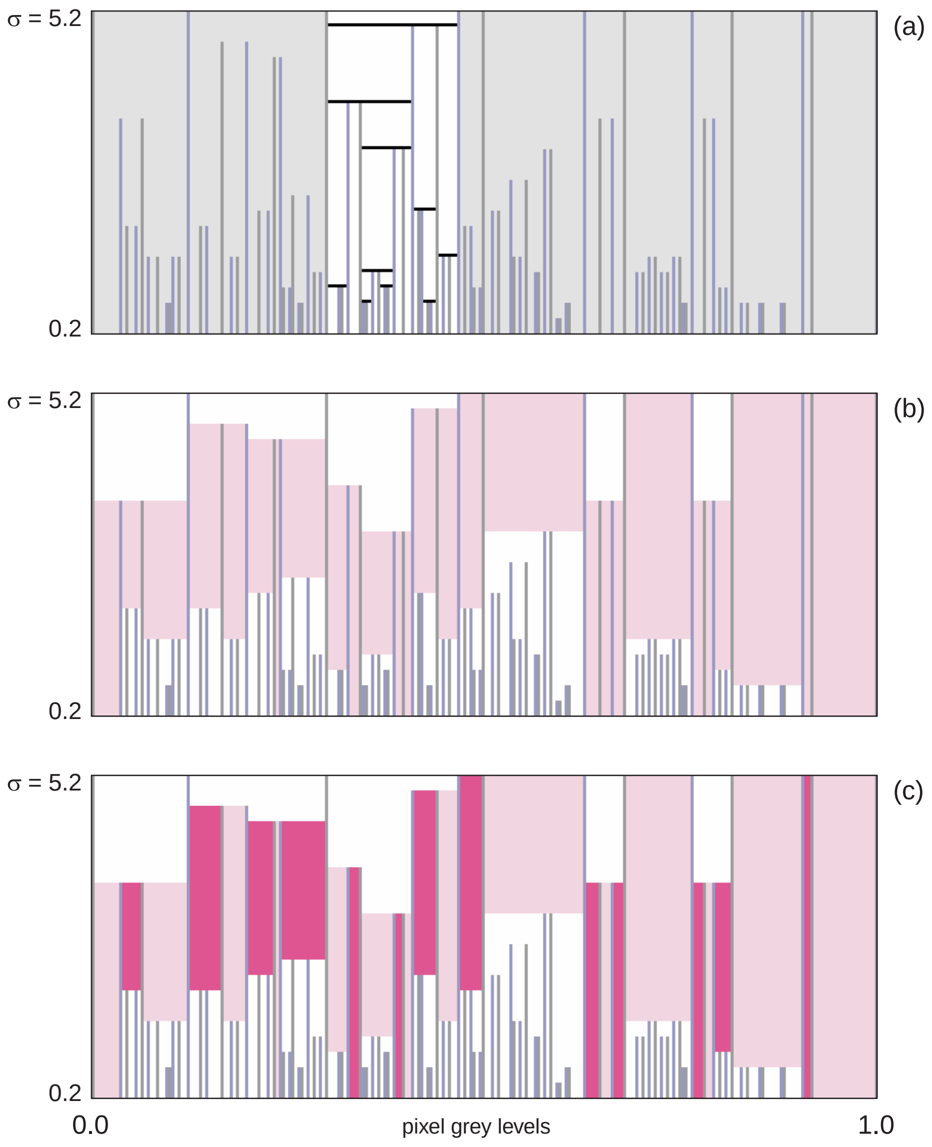


Figure 6.3: *A few images illustrating the construction of the interval tree. (a) The structure of the interval tree shown in the white region. Note how every interval has three children. (b) The maximum stability regions in the tree. (c) The maximum stability regions that contain a local maxima shown as the darker areas in the figure.*

Figure 6.4: *A sample result of the hand shading algorithm.*

by their end points. The following section will describe the details of this algorithm.

The shading algorithm requires the following parameters: 1) `arc_length` and `arc_radius`, the average arc length and radius of the strokes to be drawn, 2) `arc_orient`, the angle in which the shaded pattern will be oriented, 3) `arc_dist`, the average distance between the midpoints of the drawn arcs, 4) `arc_center`, the initial arc center, and 5) `arc_number`, the total number of arcs to be drawn. These parameters are illustrated in Figure 6.5a,b.

The algorithm begins by computing, `arc_start`, the starting location of the first arc to be drawn. This computation is done by applying a standard translation and rotation to the `arc_center`. Once the starting point is found, an arc is drawn who's length is equivalent to `arc_length`. The point at the end of the arc is labeled `arc_end` (Figure 6.5a). Next, the system is rotated slightly about `arc_end` and the direction of drawing is reversed. The rotation angle is determined by `arc_dist`.

Assuming that the distance between `arc_start` and `arc_end` is large relative to `arc_dist`, we can approximate the rotation angle, θ, by the following equation:

$$\theta = \arctan \frac{2 * \texttt{arc\_dist}}{|\texttt{arc\_start} - \texttt{arc\_end}|}$$

Figure 6.5b illustrates the parameters of this equation. Figure 6.5c,d shows the drawing of a second and third arc in the rotated system. After each iteration, the `arc_length` and θ are slightly skewed by a random amount to give a more natural feel.

6.3 Drawing and Shading Algorithms

With the two tools described above, we can describe the various algorithms that compose the drawing phase of the artistic rendering process.

As previously discussed in Chapter 3, the composition order of the the final drawing, in contrast to the computational order, is the following: 1) the shading of facial tone, 2) the drawing of edges, 3) the drawing of hair, 4) the drawing of facial features, and 5) the shading of the background region. The following section will present these operations in order.

6.3.1 Facial Tone

The facial tone layer provides a pleasing shaded quality to the final artistically rendered image. Specifically, we want to be able to create the appearance of smudged charcoal for the shaded regions of the face. The areas around the edge of the face are one such example. We use the following observations of the appearance

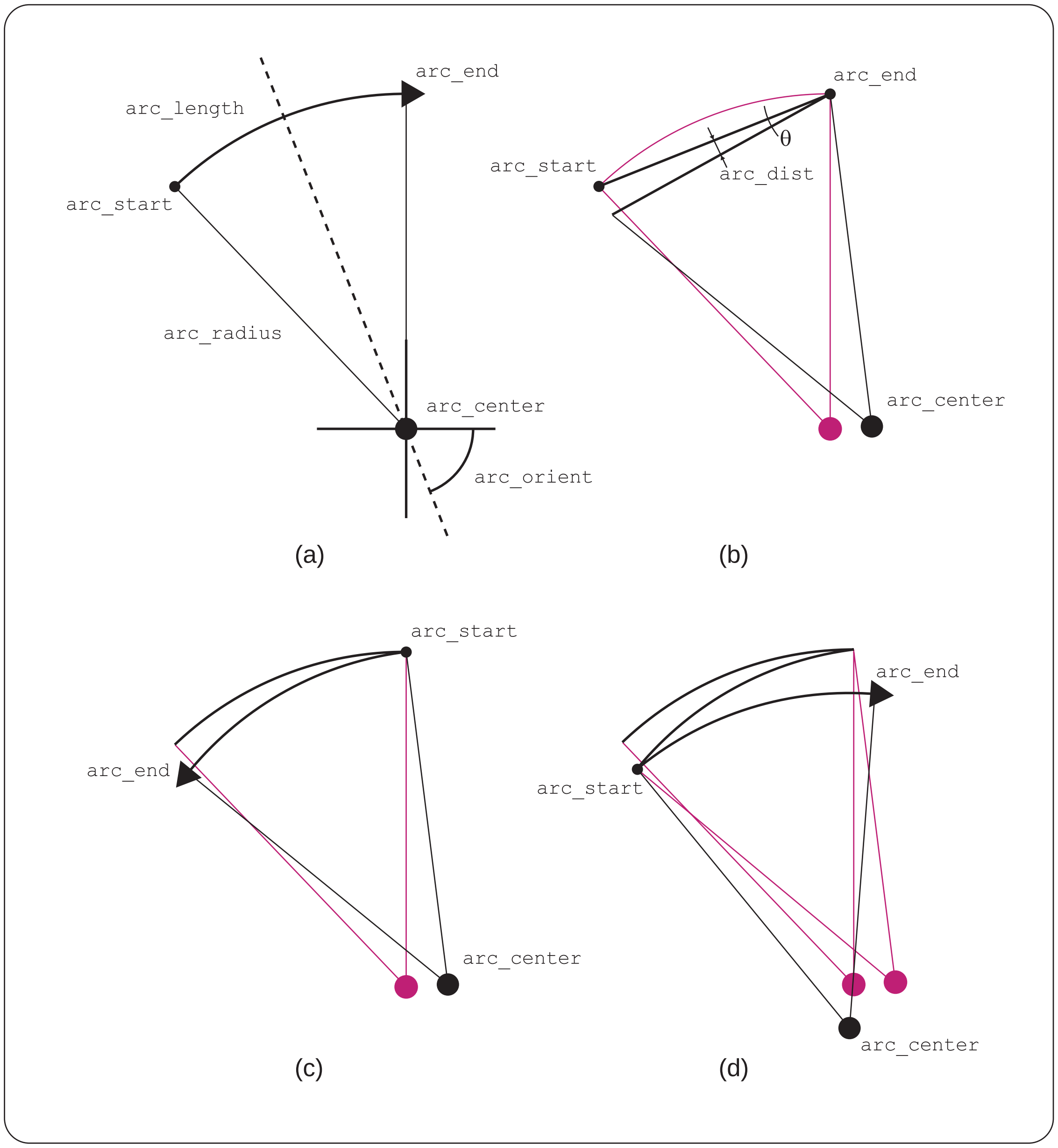


Figure 6.5: *The drawing of arcs for the hand shading algorithm. (a) The initial setup and basic parameters for the algorithm. (b) How θ is computed from* `arc_dist`. *(c), (d) A few more iterations of the shading algorithm.*

of smudged charcoal: 1) both dark and light smudging tends to create either a mostly black or mostly white solid pattern, and 2) medium smudging creates a rougher and more gritty pattern.

To simulate this behavior, we take a source grey value and blend it with a rough paper texture. In order to adhere to the observations above, we blend the two values via the following formula:

$$d = \frac{t*\mathrm{w}(s)+s*(1-\mathrm{w}(s))+n*s}{(n+1)}$$

$$\mathrm{w}(s) = \frac{1+cos(2\pi(s-1/2))}{2}$$

where s is the source grey value, t is the paper texture grey value, d is the destination grey value, $\mathrm{w}(s)$ is the weighting factor, and n is a bias factor that can weight the source image more heavily in the final blending. For our implementation, we set $n = 1$. Figure 6.6a shows a sample plot of $\mathrm{w}(s)$ where s ranges from 0 to 1. The cosine in the weighting function provides the behavior described above. Figure 6.6d shows the result of blending a paper texture (Figure 6.6b) with a grey-scale gradient image (Figure 6.6c). Figure 6.11 shows the result of applying the algorithm to the blurred image generated during the facial tone computational step.

6.3.2 Edges

The result of the edge extraction process described in Chapter 3 is a black and white image in which white pixels represent lines that the user wishes to be drawn. In general, the drawing process works by first converting the pixels into line segments and then drawing the line segments in a sketchy fashion. The main loop of the

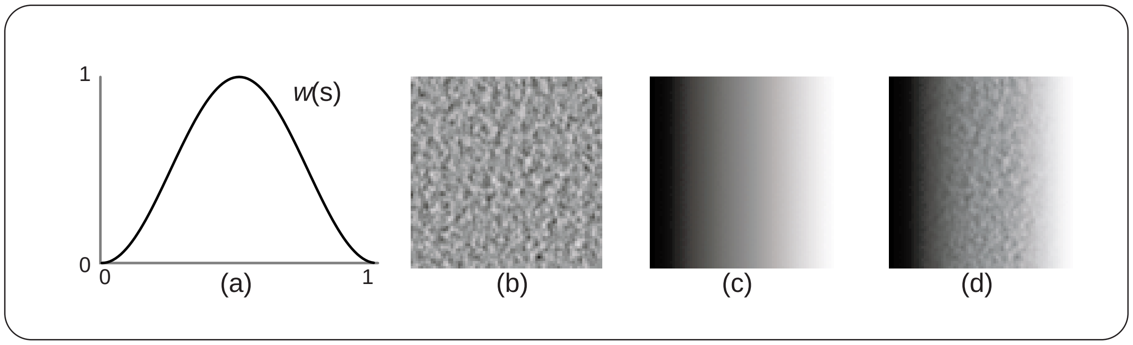


Figure 6.6: *Illustration of the generation of facial tone. (a) A plot of the function w. (b) An image of a typical paper texture. (c) A grey-scale image gradient. (d) The result of blending the paper texture with the image gradient.*

edge tracing algorithm scans through the image and searches for a pixel that lies on a line (e.g. white pixels in the black and white image). To clarify, a pixel that composes part of a line in the source image will be refered to as an *edge pixel*. When an edge pixel is found during the scanning process, the algorithm traces the line and marks the edge pixels that represent it as “visited” so that they will not be considered again. Once traced, the algorithm continues scanning through the image in search of other edge pixels. Figure 6.7a illustrates an example of an edge pixel found by the loop above.

Once an edge pixel is isolated, a recursive growing algorithm is applied to find all the connected edge pixels that lie within a prespecified radius (Figure 6.7b). For our application, we chose an initial radius of 3. Vectors pointing from the center pixel to these perimeter pixels are then created, and the average vector for each direction is computed (Figure 6.7c). In general, either one or two initial directions are found. Figure 6.7c illustrates two directions. However, if the source pixel is located at the end of a line segment, only one direction will result.

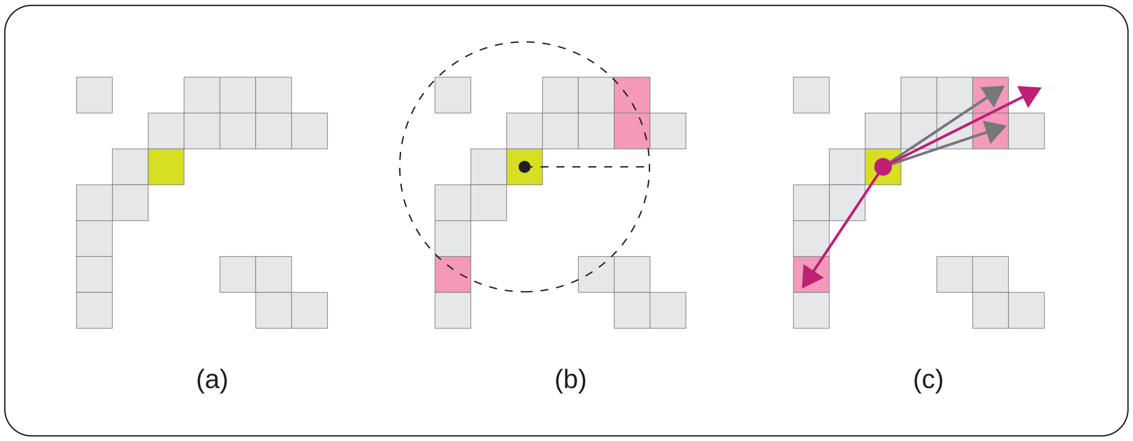


Figure 6.7: *An illustration of how to find the initial vectors for the edge tracing process. (a) A sample initial point for the edge tracing algorithm. (b) Pixel within a prespecifed radius. (c) Vectors to these points are found and averaged.*

Unfortunately, it is not always guaranteed that the initial pixel will lie on a single line. It is also possible that this pixel is either isolated in space or located at the intersection of many lines. However, in both these cases, the edge tracing algorithm works correctly. For an isolated initial pixel, the search for perimeter pixels will fail and the algorithm will return to the main loop. In the second case, vectors will be computed in erroneous directions and the line tracing algorithm will again fail and return to the main loop. Although seemingly incorrect, this failing at intersection points is not a problem because further attempts to trace each line will later be made as the main loop scans though the image.

Once an initial pixel and initial directions have been found, the line tracing can begin. Because the tracing algorithm is symmetric for each direction, we will only discuss the tracing of a line in a single direction.

The first step in the line tracing algorithm involves scaling the initial direction vector to a length of three. Next, the direction vector is added to the initial pixel location resulting in a new point that predicts the direction of the line (Figure 6.8a). The 5×5 region around the predicted point is then scanned where the closest edge pixel to the predicted point is found (Figure 6.8b). Also, during the scanning process, all edge pixels in the region are marked as “visited” to prevent backtracking and later consideration. If no edge pixels are found in the region, the algorithm returns to the main loop.

The location of the closest edge pixel to the predicted point is added to the final line segment and is also set as the new initial pixel (Figure 6.8c). With this new initial pixel, a new direction vector is computed by averaging the previous direction vector with the vector that points from the previous initial pixel to the new initial pixel (Figure 6.8c). This process is repreated until the terminating condition is met (Figures 6.8d,e). Finally, after the completion of the main loop, the individual line segments computed from the tracing algorithm are linked. This linking is done by joining lines with similar end points.

Once the edges have been vectorized, they are then drawn in a sketchy fashion. Essentially, this sketchiness is achieved by drawing slightly displaced subsets of the line segment. Specifically, the line is first broken up into a set of random overlapping segments. Random offsets are then computed for the start and end of each segment. Finally, each segment is drawn while being displaced from the original line by the offsets previously computed. After the completion of this process, the original line is drawn to emphasize the final line. Figure 6.9 compares two images drawn with and without added sketchiness.

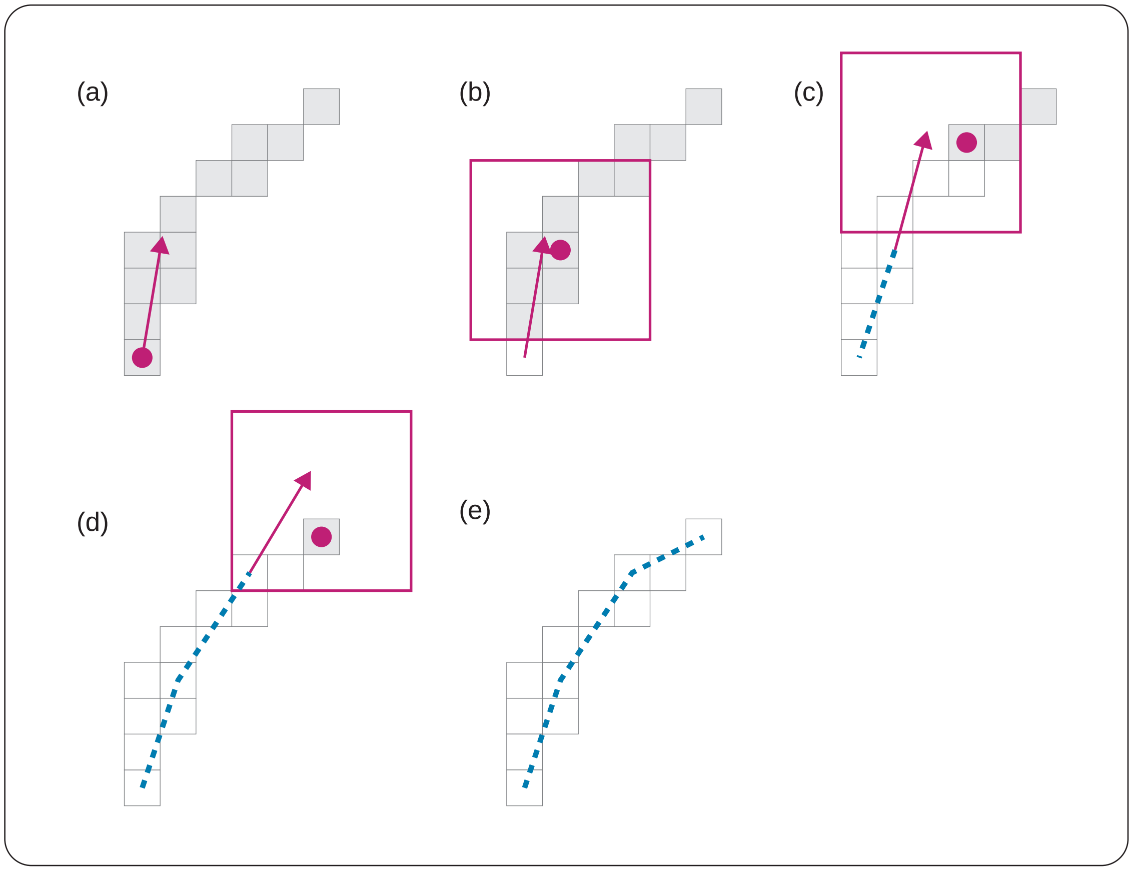


Figure 6.8: *A few diagrams illustrating the edge tracing process. (a) A sample line to be traced. The inital point and tracing directions are shown. (b) The 5×5 region around the predicted point and the closest edge pixel. (c) The closest edge pixel becomes the new inital pixel, and a new vector direction is computed. White pixels represent edge pixels marked as "visited." (d), (e) The continuation of the process. The blue dotted line is the final extracted line segment.*

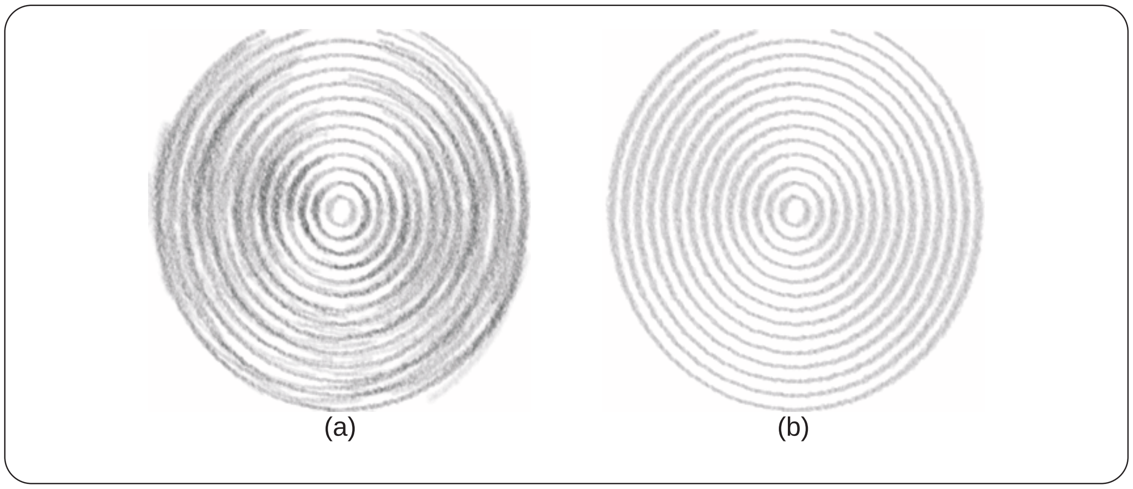


Figure 6.9: *The difference between the drawing of edges (a) with the sketchy drawing algorithm and (b) without.*

The results of the edge drawing algorithm are shown in Figure 6.12.

6.3.3 Hair

Once the orientation map from the orientation filter has been generated, the drawing of hair simply involves tracing lines along the orientation map with a user specified length and number. To do this tracing, we begin by picking a random starting position, and then growing a line in both directions based on the underlying orientation. The line is grown until a prespecified length is met or the line has grown outside of the hair region.

The actual growing algorithm works as follows. A random point in the hair region is chosen. From this point, two line segments are drawn in opposing directions. The directions of the segments are aligned to the orientation vector located at the seed point. The end points of the two segments are used as new seed points. The orientation under these points are sampled, and a new line segment is drawn from these points. However, unlike the initial condition, only one segment per seed point is drawn in an outward direction. This process is repeated until one of the terminating conditions is met.

To add a more natural feel, an angular offset is added to the value sampled from the orientation field. This offset is randomly chosen at the start of each line. For our application, we chose random offsets between $-\pi/12$ and $\pi/12$, and have found that these values increase naturalness while still preserving the general orientation of the hair. Figure 6.10 shows the results of this growing process with and without using the offset. In addition to this angular offset, the length of each hair segment is randomly altered as well.

As a final note, 500 to 1500 hair strokes are generally drawn to create the results shown in Chapter 8. The number of strokes depends greatly on the size of the hair region as well as the average length of strokes used to cover it. Furthermore, although we employ no algorithms to guarantee even coverage of strokes over the hair region, it is probabilistic given the number of strokes that are typically drawn.

The results of the hair drawing algorithm are shown in Figure 6.13.

6.3.4 Facial Features

After the facial features have been clustered, the eyes, nose and mouth are thresholded as described in Section 6.1. Each thresholded region is then shaded using the algorithm described in Section 6.2. Unlike other facial features, however, the eyebrows are not thresholded. Instead, these regions are filled using only the hand shading algorithm. We chose to use a larger spacing between the strokes and to

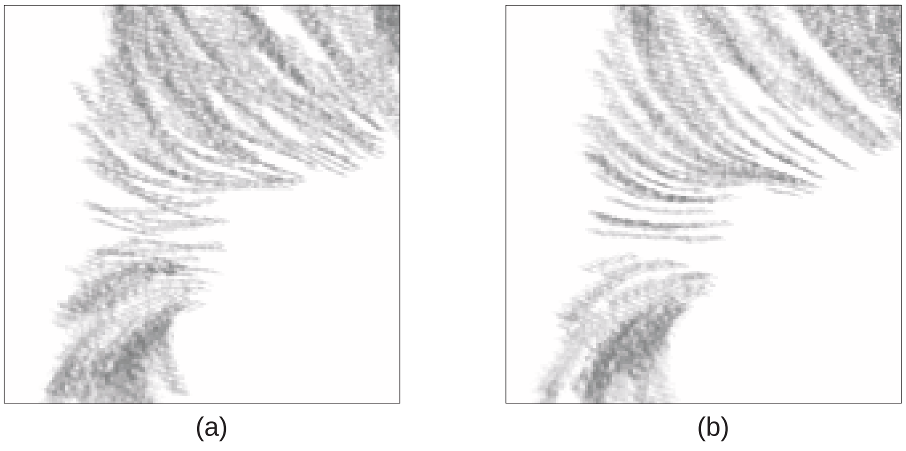


Figure 6.10: *A sample hair rendering showing the difference between the drawing of hair (a) with random offsets and (b) without.*

orient the shading direction of each eyebrow in opposing directions. These parameters give the eyebrows a more directed appearance.

The results of shading the facial features are shown in Figure 6.14.

6.3.5 Background

The final and most straight forward drawing and shading algorithm is the one for the background region. This shading simply involves taking the segmented region and applying the hand-shading algorithm to it. Figure 6.15 shows a sample result.

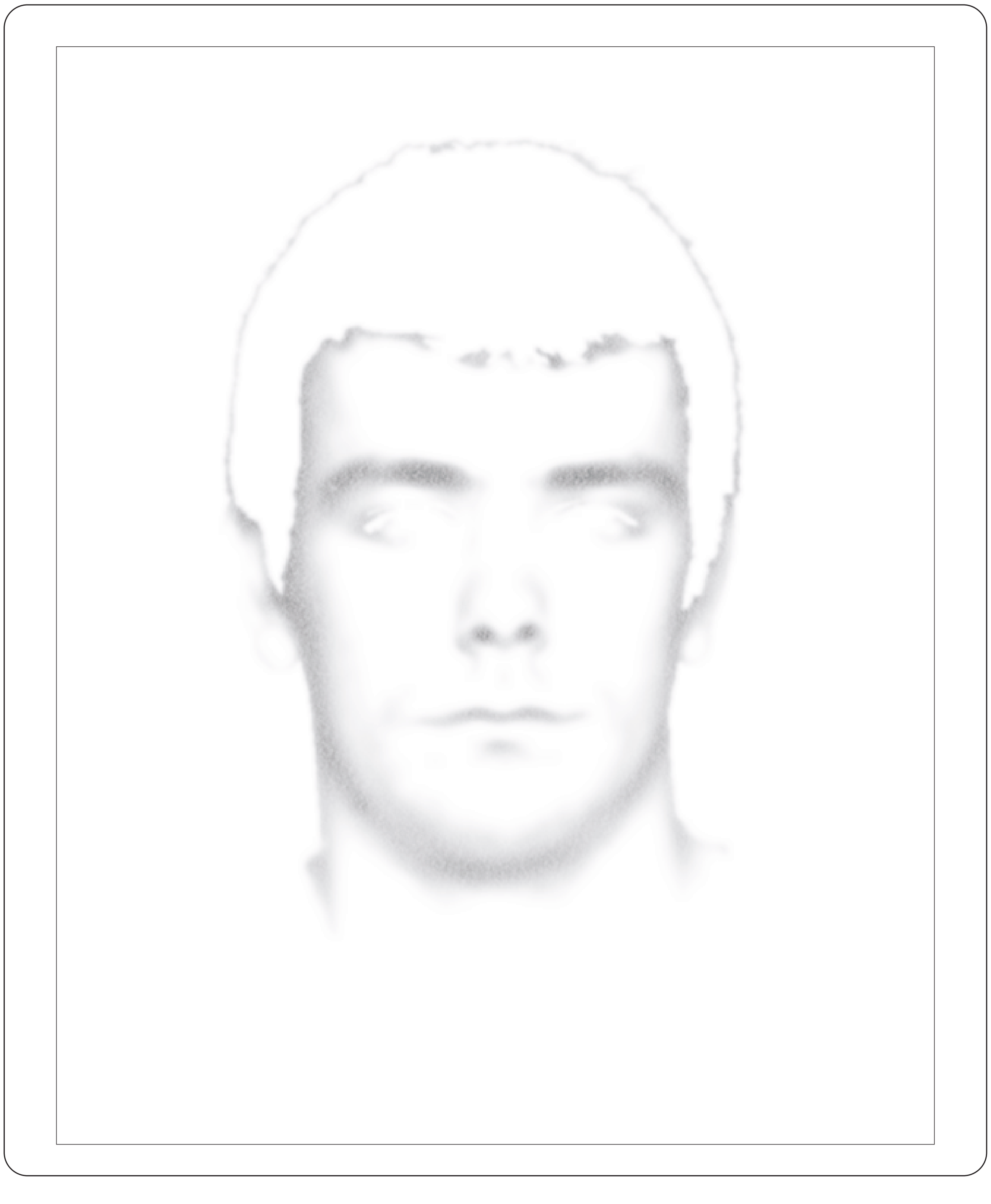


Figure 6.11: *A sample result of shading facial tone.*

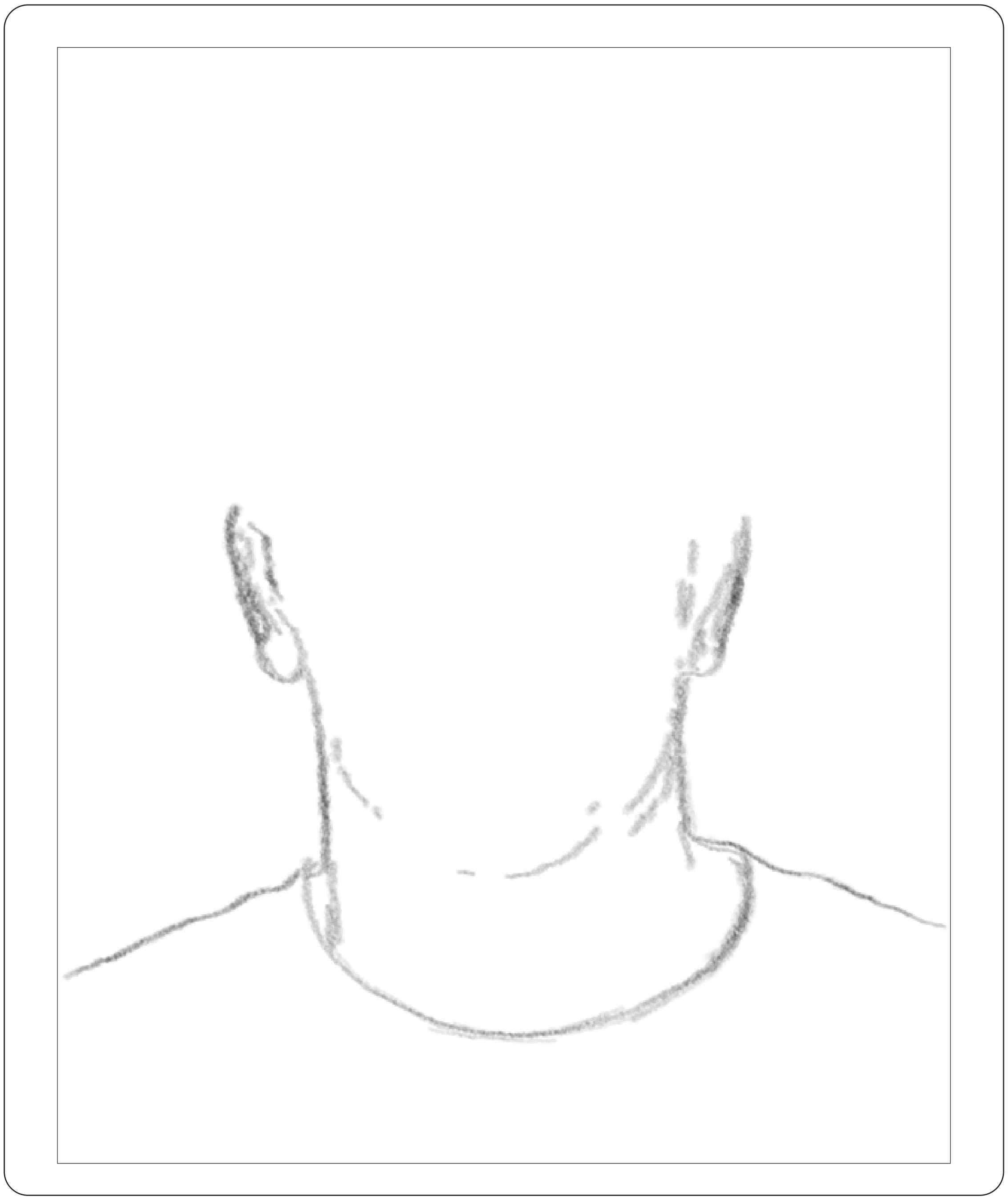


Figure 6.12: *A sample result of tracing and drawing the edges.*

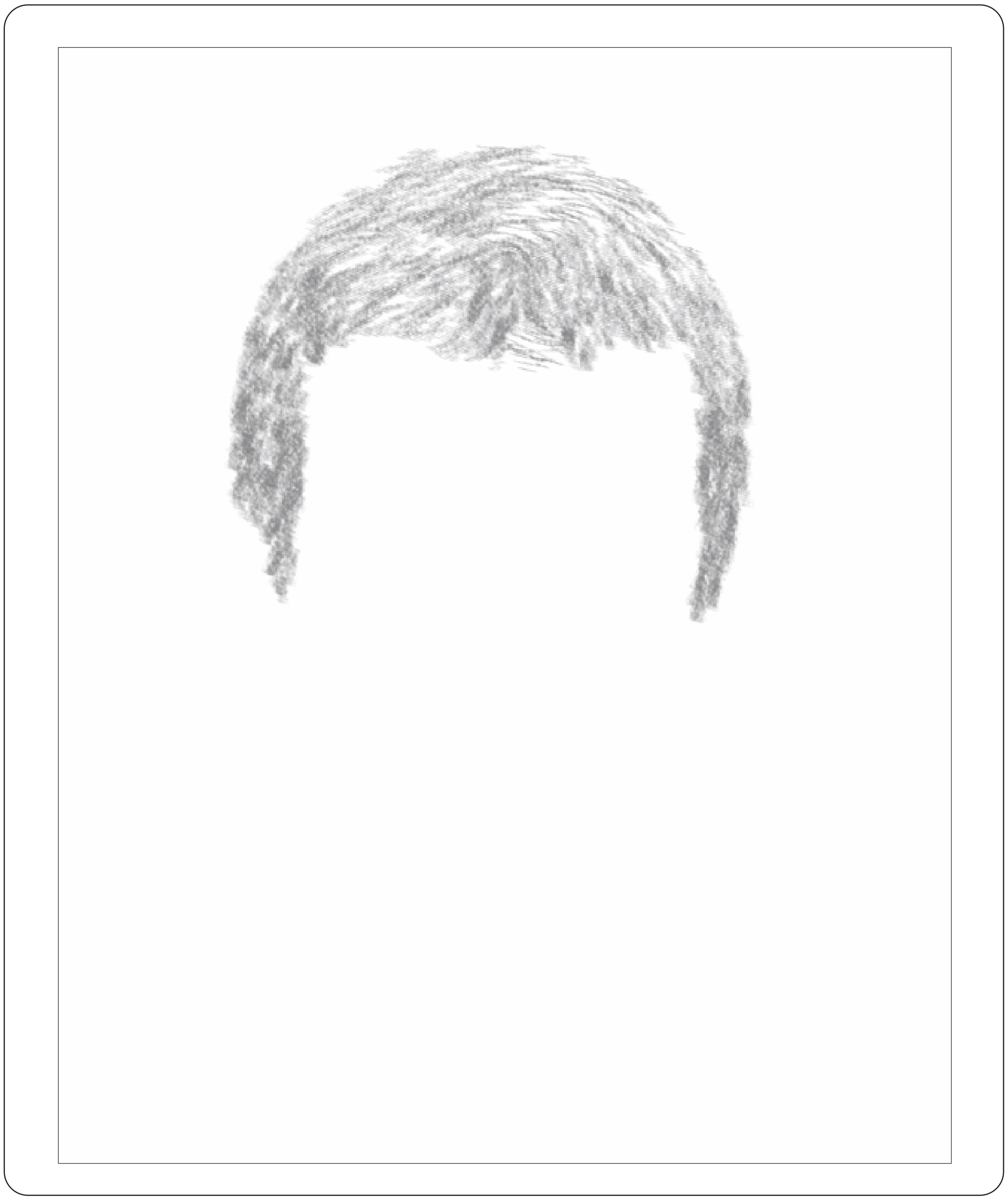


Figure 6.13: *A sample result of drawing hair.*

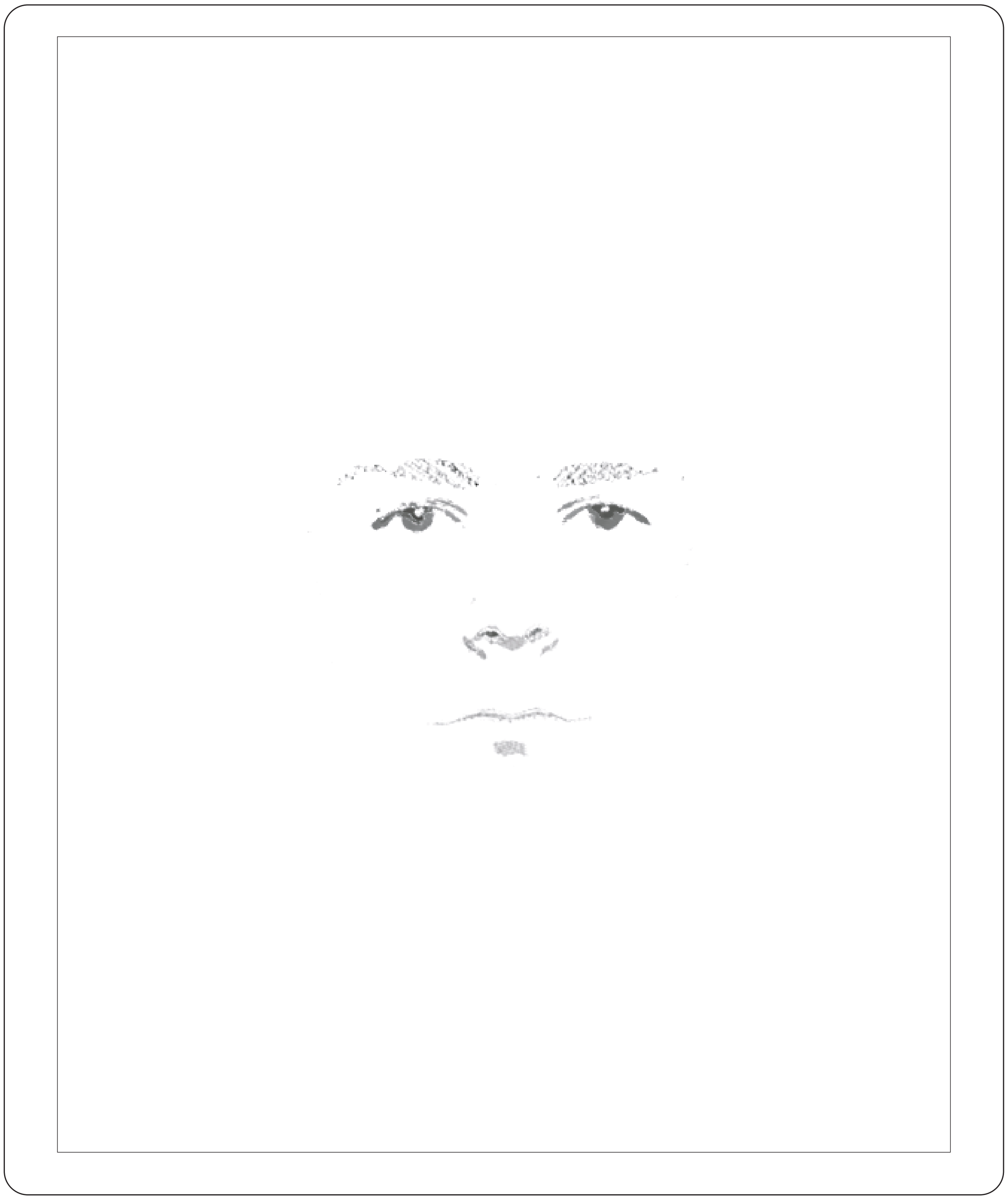


Figure 6.14: *A sample result of the drawing of facial features.*

Figure 6.15: *A sample result of shading in the background region.*

Chapter 7

Results

This chapter illustrates sample results of the artistic rendering system that we have developed in this thesis. The source image sizes ranges from 500×600 to 600×800 pixels, and the processing time for the images range from 10 to 20 minutes, depending on the amount of user interaction involved.

We demonstrate that our artistic rendering system can create convincing charcoal sketches from portrait photographs. More importantly, however, is the fact that in the resulting sketches, our system is able to capture small subtleties from the source portrait that define the expression and the essence of the subject.

Although our artistic rendering system works for a wide range of portraits, it has limitations. Glasses or any other occluding objects will pose a problem for our system. These objects can cause errors in the facial feature clustering algorithm. Darker skin complexions as well as dark spots caused by birthmarks or dimples can also throw off the facial feature clustering algorithm. Curly hair is also problematic because of the manner in which we apply an orientation filter to find and draw the hair region.

Figure 7.1 shows a sample portrait and Figure 7.2 shows the result of artistically rendering this image. Note that this drawing was rendered without the background shading step. The choice of shading the background is left to the user. Figures 7.3 and 7.4 show another sample set of images. Figures 7.5 and 7.6 show the same subject from the previous set of image smiling. Notice how the bottom lip of 7.6 is washed out in the final image. This is due to the fact that the lips in the source image are thin and close in color to the subject's skin tone. Furthermore, the specular highlight on the lip throws off the facial clustering algorithm as well. Figures 7.7 and 7.8 illustrate how our system can handle facial hair. For this image, the removal of edges was a more difficult process because of the shirt pattern on the subject. A final set of sample images are shown in Figures 7.9 and 7.10.

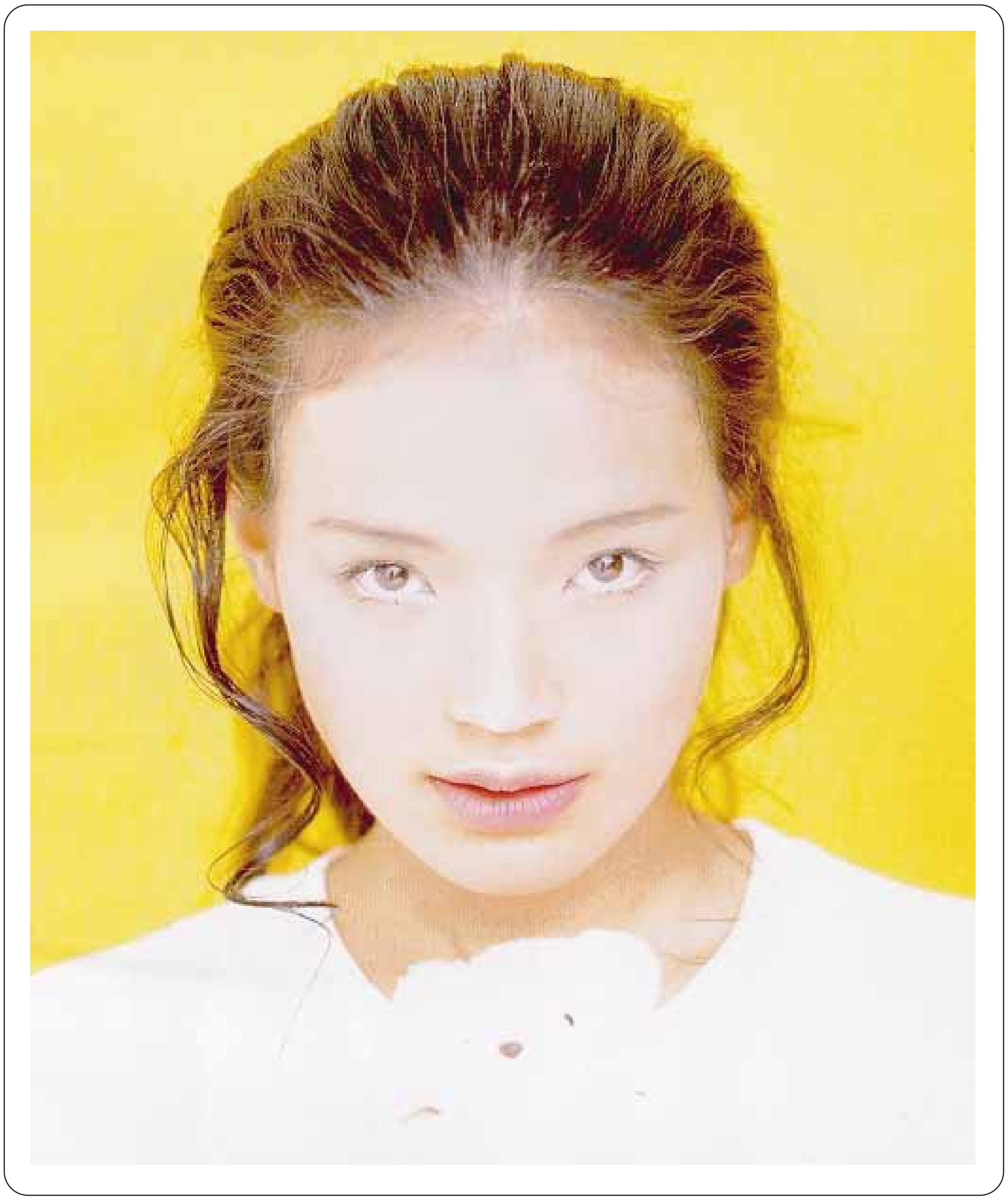


Figure 7.1: *Subject A: Portrait Photograph*

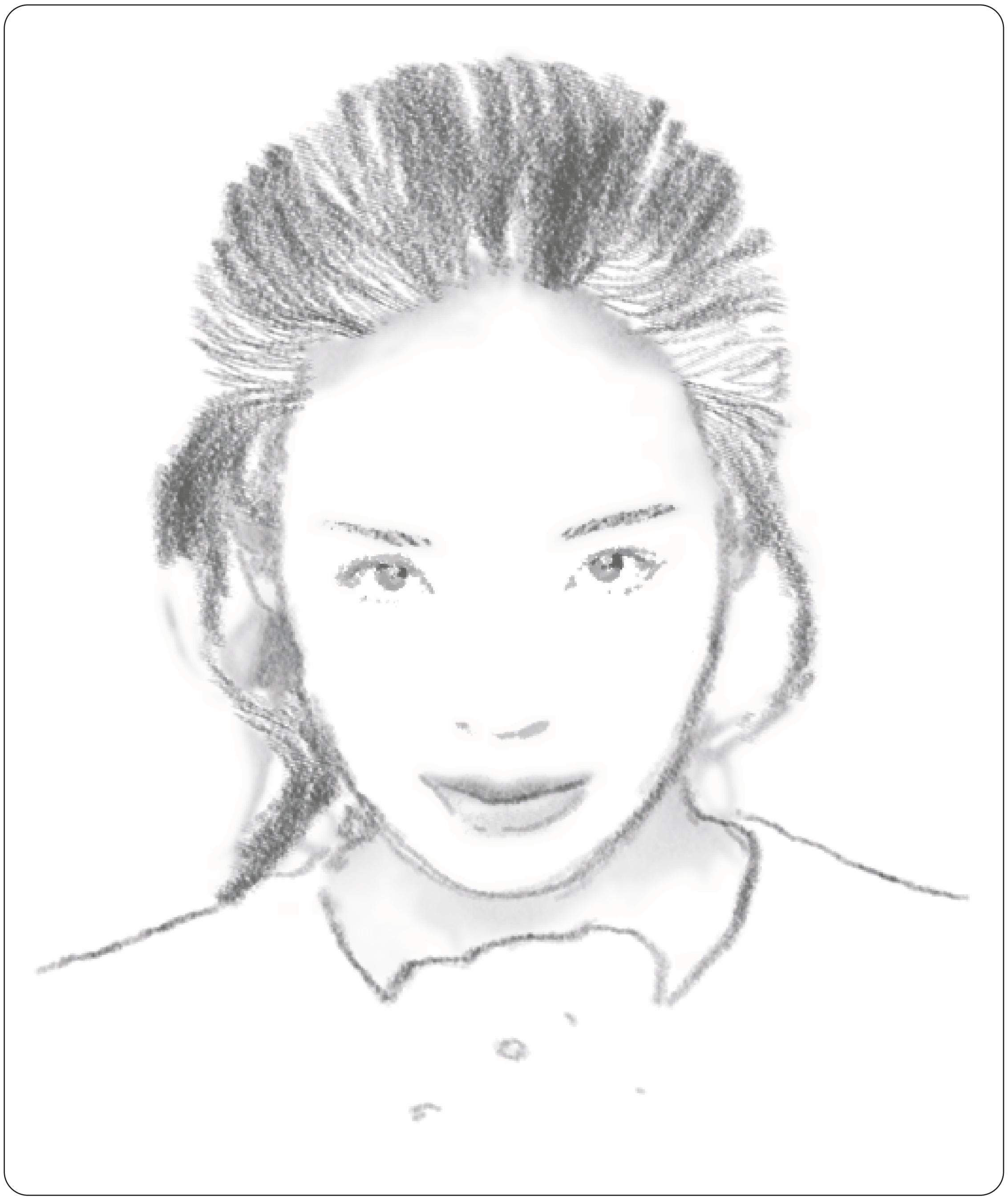


Figure 7.2: *Subject A: Artistic Rendering*

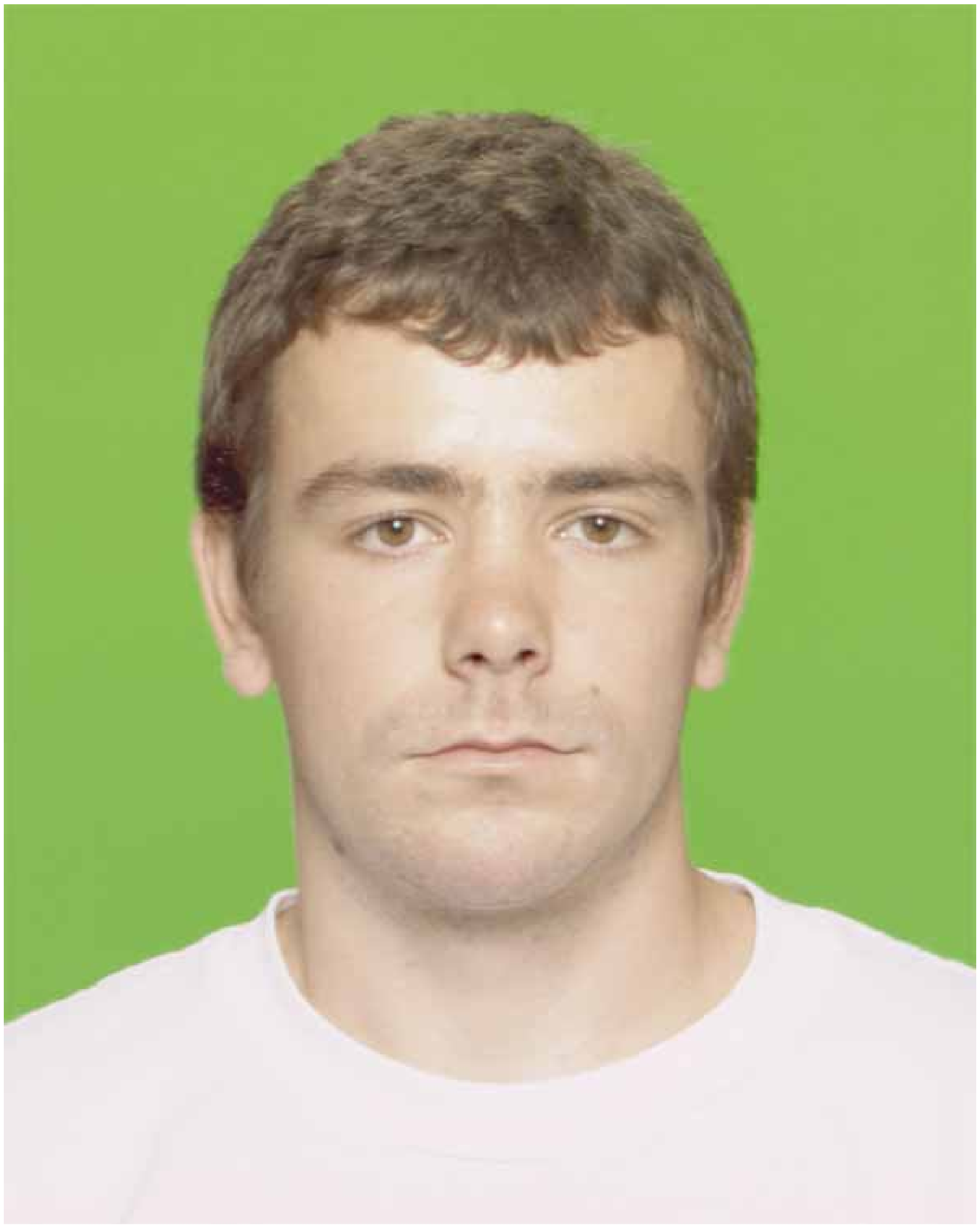


Figure 7.3: *Subject B: Portrait Photograph*

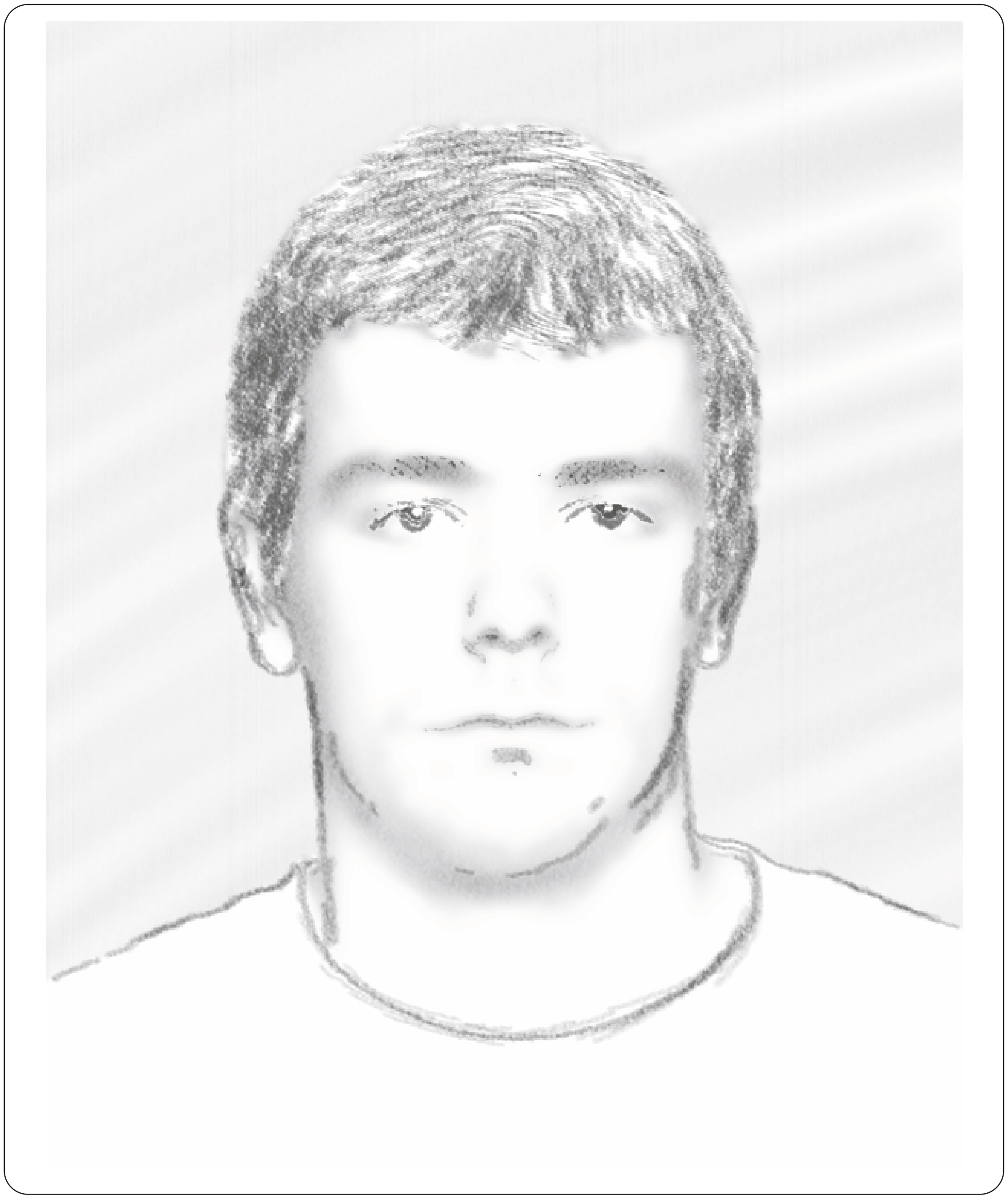


Figure 7.4: *Subject B: Artistic Rendering*

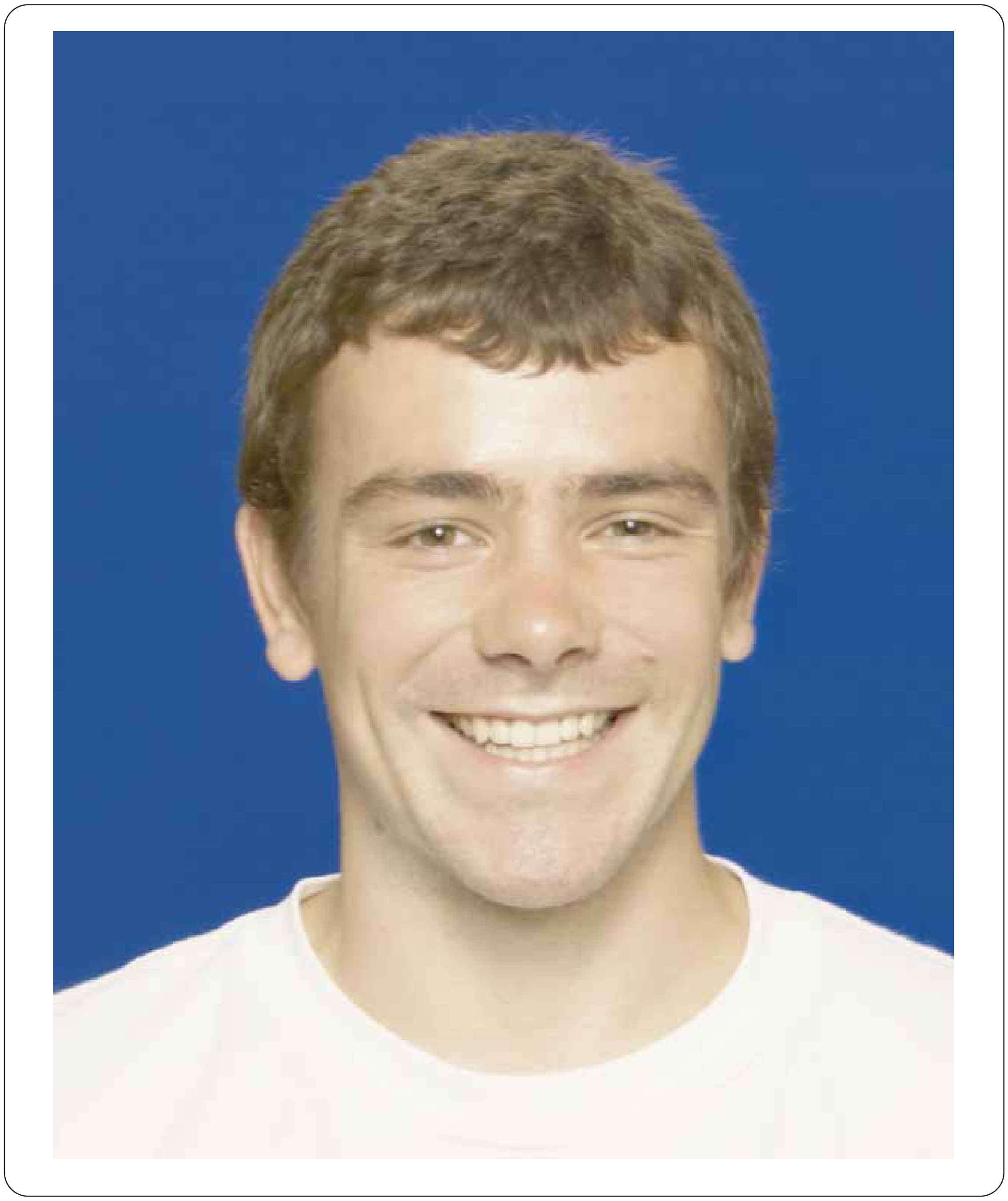


Figure 7.5: *Subject B: Portrait Photograph (smiling)*

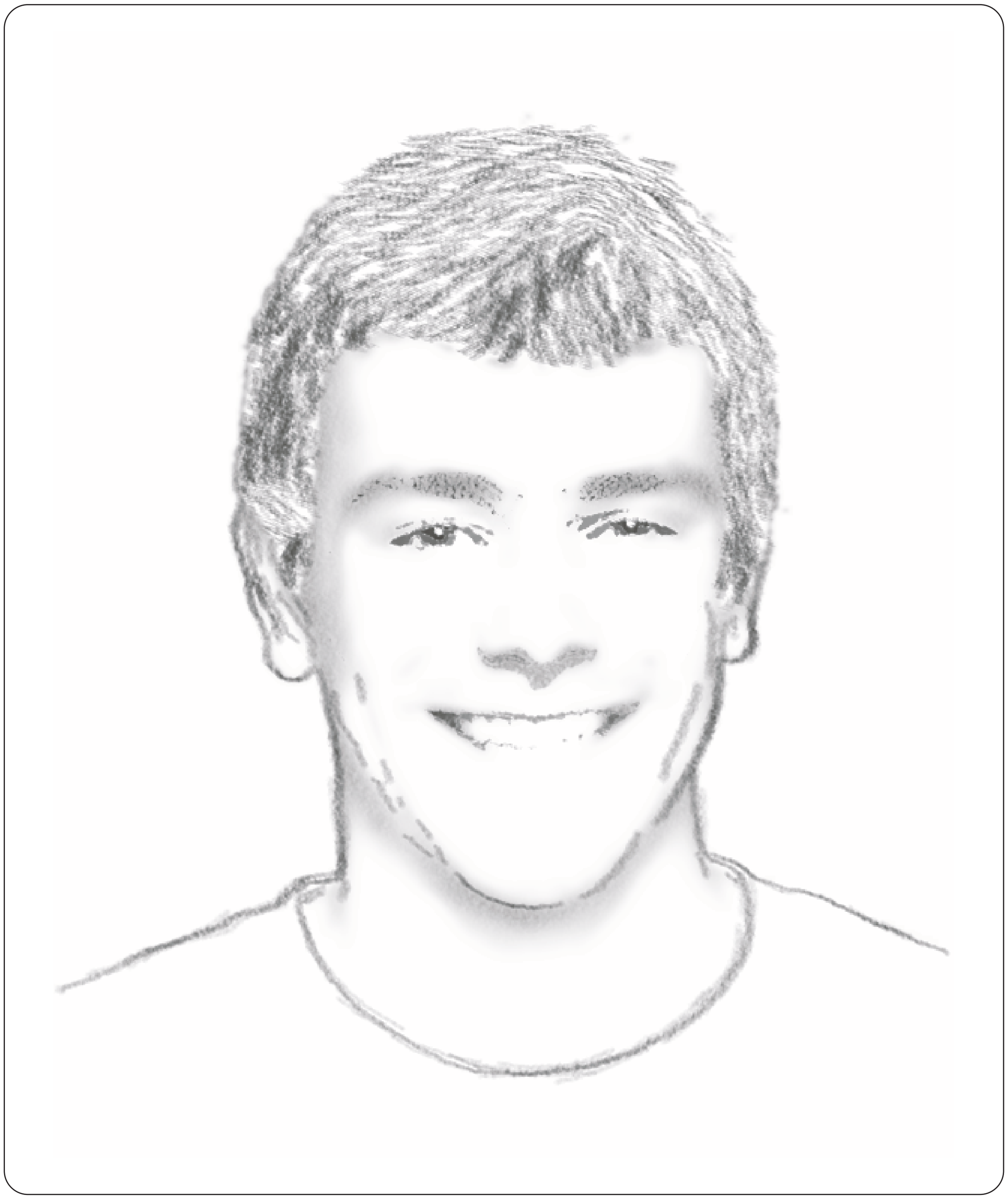


Figure 7.6: *Subject B: Artistic Rendering (smiling)*

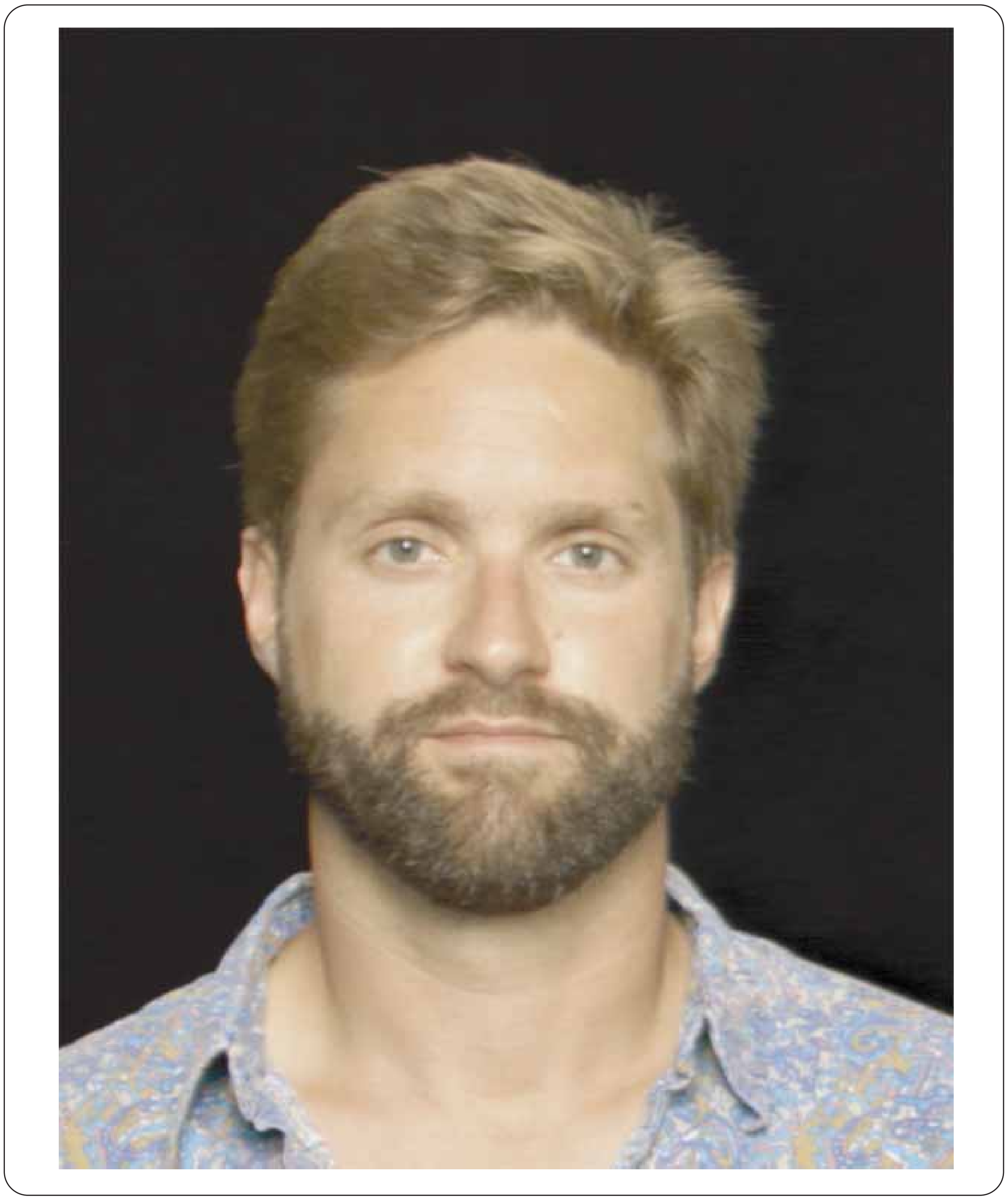


Figure 7.7: *Subject C: Portrait Photograph*

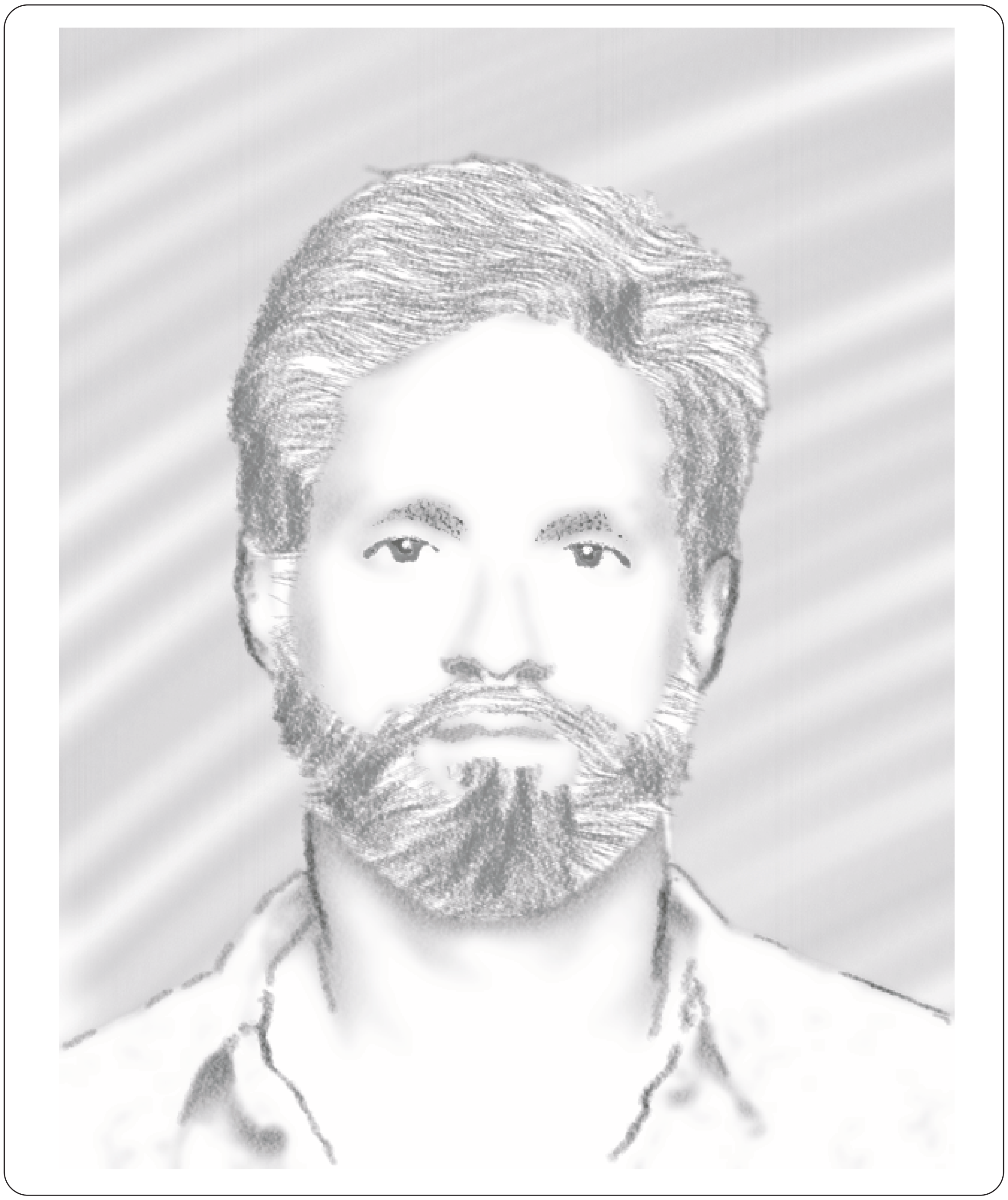


Figure 7.8: *Subject C: Artistic Rendering*

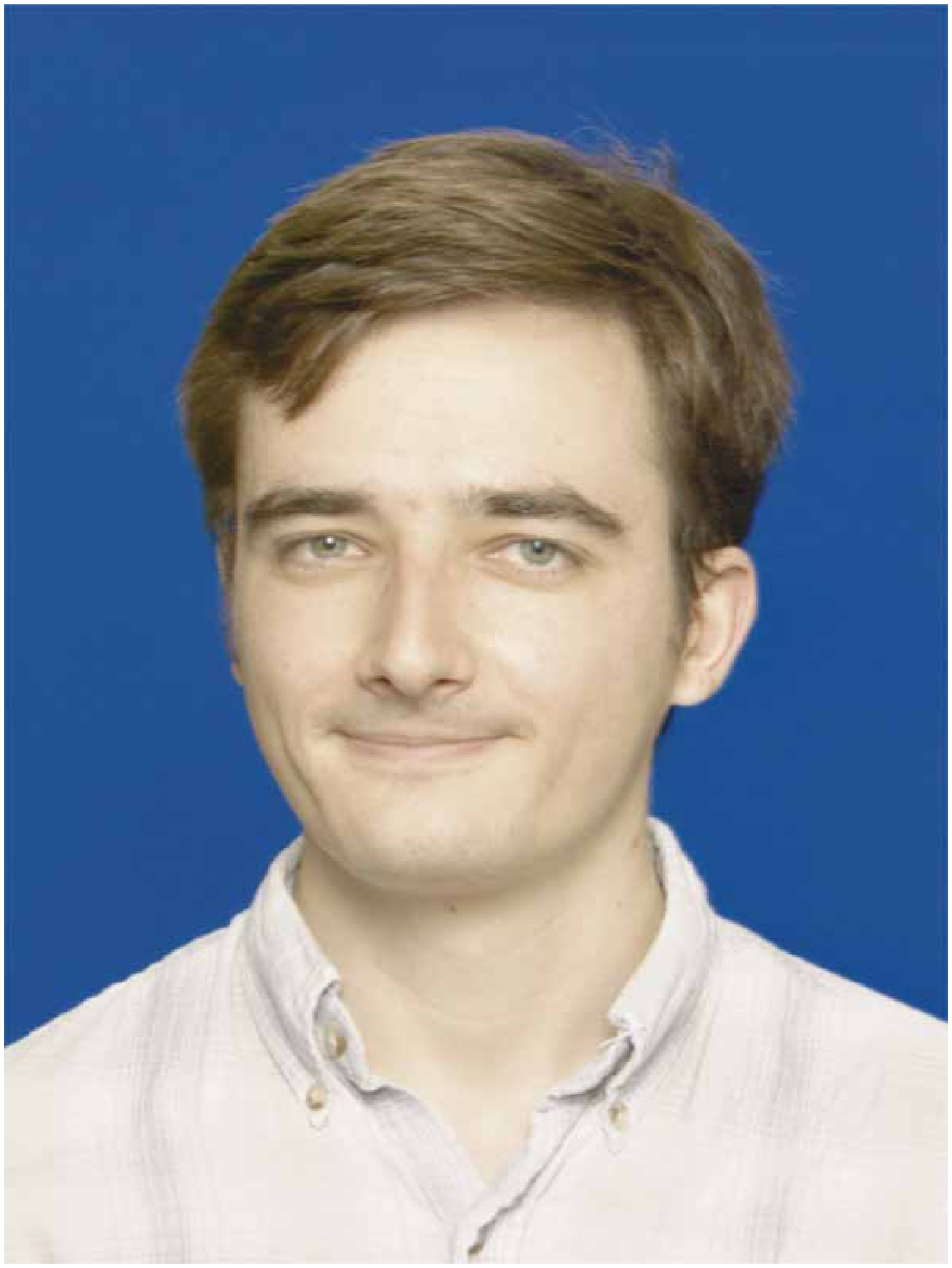


Figure 7.9: *Subject D: Portrait Photograph*

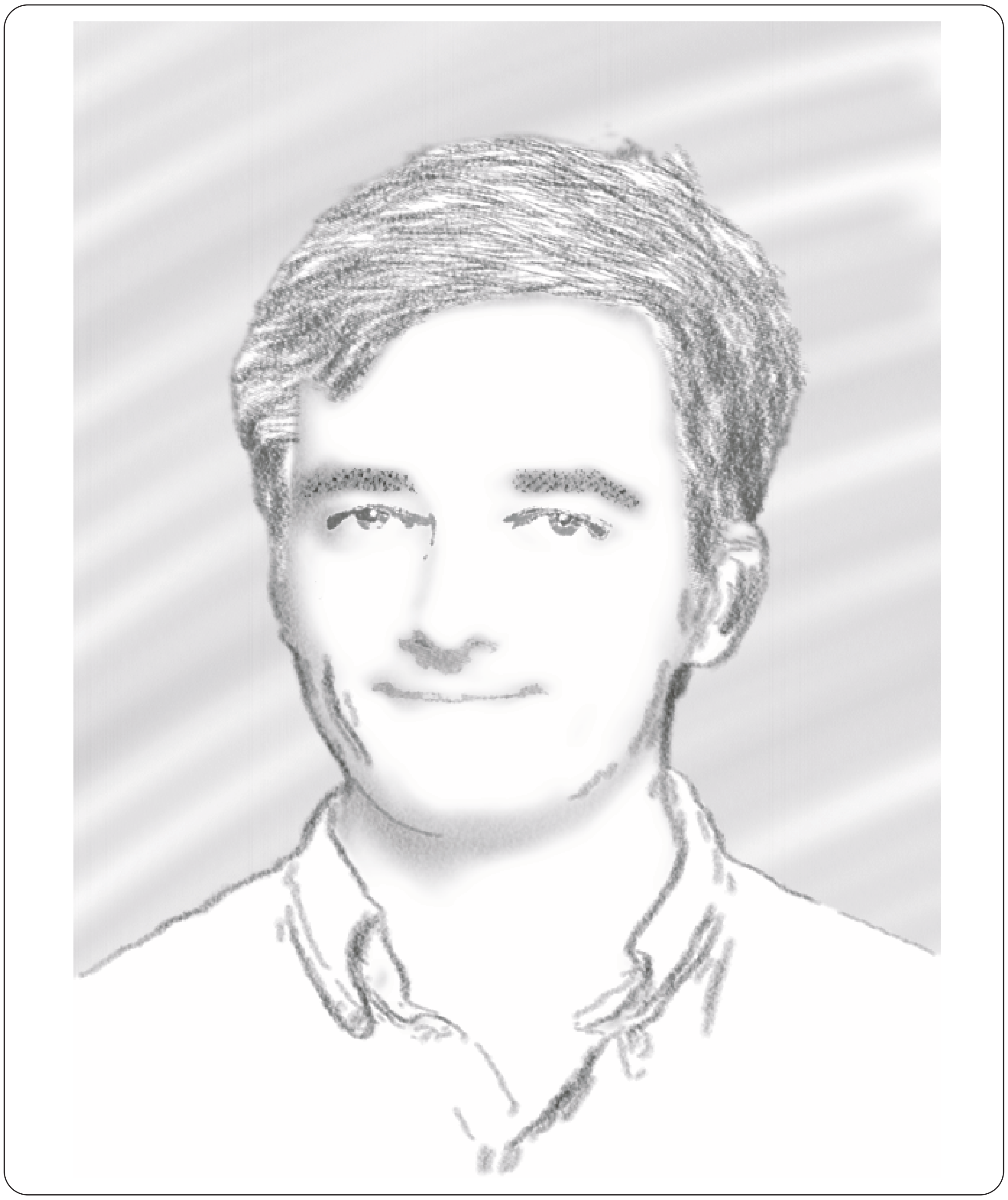


Figure 7.10: *Subject D: Artistic Rendering*

Chapter 8

Conclusion and Future Work

In the past decade, artistic rendering has become an exciting research area in the computer graphics community. During this time, a wealth of artistic rendering techniques have been explored that vary both in their source data assumptions and their target medium types. In this thesis, we introduced a semi-automatic artistic rendering system for generating charcoal-style drawings from portrait photographs. We believe our system differs from previous work in the following manner. First, our system is the first to primarily focus on generating charcoal style drawings. Second, it is focused on rendering only portrait photographs. Although seemingly detrimental, this restriction allows us to make assumptions during the rendering process to enable a higher quality final image.

In Chapter 3, we reviewed the overall artistic rendering process for converting from portrait photographs to artistic drawings. In summary, the process is broken down into the following five main areas: 1) the background area, 2) the hair, 3) the edges and lines, 4) the facial features, and 5) the facial tone. Chapters 4 though 6 then reviewed the major components of each of these steps in greater detail. Chapter 7 presented the final results of our artistic rendering process and illustrated how our system can produce convincing charcoal sketches that convey the essence of their source portraits.

The most significant area of improvement for our artistic rendering system is to improve robustness. As discussed in Chapter 7, our system is limited in the scope of the faces that it can render. For example, glasses pose a problem for our system as do people with darker skin complexions. Dark areas on the face such as those caused by birthmarks or dimples can also throw off the facial feature clustering algorithm. Curly hair is also problematic.

The artistic rendering system can also be improved by automating the user interactive processes. Currently, the user is required to manually select the regions of the image that represent the background, hair, face, and edges. If these processes could be automated, then our artistic rendering system could be used with little or no instruction or implemented as a "plug-in" module for a photo manipulation program such as Adobe Photoshop [24]. For instance, template matching techniques could be used to locate major features of the face. Furthermore, in conjunction with the information found from the template matching, texture segmentation techniques could be applied to automatically isolate hair regions of the portrait.

However, the isolation of edges in the source image may be a more difficult task to automate because choosing which edges are to be drawn in the final image is relatively subjective. Template matching methods can also be explored here.

Finally, additional metrics can be explored for automatically choosing parameters such as hair color and the pen pressure for the drawing of facial features.

Currently, the Hough orientation filter takes roughly 30 seconds to run on a dual-processor 300MHz Pentium II machine for a typical hair region. A few optimizations have already been implemented for the Hough orientation filter, but as discussed in Section 2.2.1 there are still possibilities for improvement. Most notably, the results of Kiryati [27] show potential for a significant performance increase via the Probabilistic Hough Transform. In addition to increasing performance, a study comparing the Hough orientation filter to other orientation filters would be useful.

Currently, we use the same drawing algorithm for the eyes, nose, and mouth. Specialized drawing algorithms for each of these features could ideally produce a more attractive final drawing. Also, we randomly render hair with strokes to probabilistically fill the entire hair region. An algorithm to ensure even coverage would be beneficial.

In summary, our system for semi-automatically generating a charcoal style drawing from a source portrait photograph is the first content specific artistic rendering system for rendering portrait photographs. Although there is much work to be done before this system can be widely applied, we hope to have provided a solid foundation to the problem of artistically rendering portrait photographs.

Bibliography

[1] A. Appel. The notion of quantitative invisibility and the machine rendering of solids. In *Proceedings of the ACM National Conference*, pages 387–393, 1967.

[2] D. H. Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2):111–122, 1981.

[3] Teresa W. Bleser, John L. Sibert, and J. Patrick McGee. Charcoal sketching: Returning control to the artist. *ACM Transactions on Graphics*, 7(1):76–81, 1988.

[4] R. Bulot, J.-M. Boi, J. Sequeira, and M. Caprioglio. Contour segmentation using hough transform. In *International Conference on Image Processing*, volume 3, pages 583–586, 1996.

[5] John Francis Canny. Finding edges and lines in images. Technical report, MIT Artifical Intelligence Labratory, 1983.

[6] Rama Chellappa, Charles L. Wilson, and Saad Sirohey. Human and machine recognition of faces: A survey. *Proceedings of the IEEE*, 83(5):705–741, 1995.

[7] Richard Coutts and Donald P. Greenberg. Rendering with streamlines. In *Computer Graphics Proceedings, Annual Conference Series, Visual Proceedings*, page 188, 1997.

[8] Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. Computer-generated watercolor. In *Computer Graphics Proceedings, Annual Conference Series*, pages 421–430, 1997.

[9] Viewpoint Datalabs. Liveart98. http://www.viewpoint.com.

[10] E. R. Davies. Image space transforms for detecting straight edges in industrial images. *Pattern Recognition Letters*, 4:185–192, 1986.

[11] Debra Dooley and Michael F. Cohen. Automatic illustration of 3d geometric models: Lines. *Computer Graphics*, 24(2):77–82, 1990.

[12] Debra Dooley and Michael F. Cohen. Automatic illustration of 3d geometric models: Surfaces. In *Proceedings of Visualization '90*, pages 307–314, 1990.

[13] R. D. Duda and P. E. Hart. Use of the hough transform to detect lines and curves in pictures. *Communications of the Association of Computing Machinery*, 15:11–15, 1972.

[14] Gershon Elber. Line art rendering via a coverage of isoparametric curves. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):231–239, 1995.

[15] William T. Freeman and Edward H. Adelson. The design and use of steerable filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(9):891–906, 1991.

[16] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1992.

[17] Amy Gooch, Bruce Gooch, Peter Shirly, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Computer Graphics Proceedings, Annual Conference Series*, pages 447–452, 1998.

[18] David S. Goodsell and Arthur J. Olson. Molecular illustration in black and white. *Journal of Molecular Graphics*, 10:235–240, 1992.

[19] Paul Haeberli. Paint by numbers: Abstract image representations. *Computer Graphics*, 24(4):207–214, 1990.

[20] Aaron Hertzmann. Painterly rendering with curved brush strokes of multiple sizes. In *Computer Graphics Proceedings, Annual Conference Series*, pages 453–460, 1998.

[21] P. V. C. Hough. A method and means for recognizing complex patterns. U. S. Patent 3,069,654, 1962.

[22] Siu Chi Hsu and Irene H. H. Lee. Drawing and animation using skeletal strokes. In *Computer Graphics Proceedings, Annual Conference Series*, pages 109–118, 1994.

[23] J. Illingworth and J. Kittler. A survey of the hough transform. *Computer Vision, Graphics, and Image Processing*, 44:87–116, 1988.

[24] Adobe Systems Incorporated. Photoshop. http://www.adobe.com.

[25] Bernd Jähne. *Digital Image Processing.* Springer-Verlag, New York / Berlin, 3 edition, 1995.

[26] Ramesh Jain, Rangachar Kasturi, and Brian G. Schunck. *Machine Vision.* McGraw-Hill, Inc., New York, 1995.

[27] N. Kiryati, Y. Eldar, and A. M. Bruckstein. A probabilitic hough transform. *Pattern Recognition*, 24(4):303–316, 1991.

[28] H. Knutsson. Texture analysis using two-dimensional quadrature filters. In *IEEE Workshop Comp. Arch. Patt. Anal. Im. Dat. Base Man.*, 1983.

[29] Bruce Land and Jonathan Alferness. Curvature-based drawings for 3-d polygonal objects. Cornell Theory Center, Cornell University, Ithaca, NY.

[30] John Lansdown and Simon Schofield. Expressive rendering: A review of non-photorealistic techniques. *IEEE Computer Graphics and Applications*, 15(3), 1995.

[31] V. F. Leavers. *Shape Detection in Computer Vision Using the Hough Transform.* Springer-Verlag, New York / Berlin, 1992.

[32] Wolfgang Leister. Computer generated copper plates. *Computer Graphics Forum*, 13(1):69–77, 1994.

[33] Young Won Lim and Sang Uk Lee. On the color image segmentation algorithm based on the thresholding and the fuzzy c-means techniques. *Pattern Recognition*, 23(9):935–952, 1990.

[34] Peter Litwinowicz. Processing images and video for an impressionist effect. In *Computer Graphics Proceedings, Annual Conference Series*, pages 407–414, 1997.

[35] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-time nonphotorealistic rendering. In *Computer Graphics Proceedings, Annual Conference Series*, pages 415–420, 1997.

[36] Barbara J. Meier. Painterly rendering for animation. In *Computer Graphics Proceedings, Annual Conference Series*, pages 477–484, 1996.

[37] MetaCreations. Painter. http://www.metacreations.com.

[38] Nikhil R. Pal and Sankar K. Pal. A review on image segmentation technqiues. *Pattern Recognition*, 26(9):1277–1294, 1993.

[39] Rosalind W. Picard and Monika Gorkani. Finding perceptually dominant orientations in natural textures. *Spatial Vision*, 8(2):221–253, 1994.

[40] Yachin Pnueli and Alfred M. Bruckstein. Digiürer - a digital engraving system. *The Visual Computer*, 10:277–292, 1994.

[41] Charles A. Poynton. Frequently asked questions about color, 1997. www.inforamp.net/∼poynton.

[42] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. *Computer Graphics*, 24(4):197–206, 1990.

[43] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive pen-and-ink illustration. In *Computer Graphics Proceedings, Annual Conference Series*, pages 101–108, 1994.

[44] Michael P. Salisbury, Michael T. Wong, John F. Hughes, and David H. Salesin. Orientable textures for image-based pen-and-ink illustration. In *Computer Graphics Proceedings, Annual Conference Series*, pages 401–406, 1997.

[45] Mike Salisbury, Corin Anderson, Dani Lischinski, and David H. Salesin. Scale-dependent reproduction of pen-and-ink illustrations. In *Computer Graphics Proceedings, Annual Conference Series*, pages 461–468, 1996.

[46] Ashok Samal and Prasana A. Iyengar. Automatic recognition and analysis of human faces and facial expressions: A survey. *Pattern Recognition*, 25(1):65–77, 1992.

[47] Alex Sherstinsky and Rosalind W. Picard. M-lattice: A novel non-linear dynamical system and its application to halftoning. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 2, pages II/565–II/568, 1994.

[48] Alex Sherstinsky and Rosalind W. Picard. Orientation-sensitive image processing with m-lattice – a novel non-linear dynamical system. In *IEEE International Conference on Image Processing*, volume 3, pages 152–156, 1994.

[49] Alex Sherstinsky and Rosalind W. Picard. Color halftoning with m-lattice. In *IEEE International Conference on Image Processing*, volume 2, pages 335–338, 1995.

[50] M. Shiono. Comparison experimetns of three kinds of table look-up methods for hough transform computation. *Trans. Inst. Electron. Inform. Commun. Eng. D-11, Japan*, 72(6):963–966, 1989.

[51] Steve Strassman. Hairy brushes. *Computer Graphics*, 24(4):225–232, 1986.

[52] Shimon Ullman and Whitman Richards, editors. *Image Understanding 1984*. Ablex Publishing Corporation, Norwood, New Jersey, 1984. Chapter 3, Scale Space Filtering: A new Approach to Multi-Scale Description, by Andrew P. Witkin.

[53] Luiz Velho and Jonas de Miranda Gomes. Digital halftoning with space filling curves. *Computer Graphics*, 25(4):81–90, 1991.

[54] Vincent Ward. What dreams may come. Polygram Filmed Entertainment, 1998.

[55] Georges Winkenbach and David H. Salesin. Computer-generated pen-and-ink illustration. In *Computer Graphics Proceedings, Annual Conference Series*, pages 91–100, 1994.

[56] Georges Winkenbach and David H. Salesin. Rendering parametric surfaces in pen and ink. In *Computer Graphics Proceedings, Annual Conference Series*, pages 469–476, 1996.

[57] L. Xu, E. Oja, and P. Kultanen. A new curve detection method: Randomized hough transform (rht). *Pattern Recognition Letters*, 11:331–338, 1990.

[58] Robert C. Zeleznik, Kenneth P. Herndon, and John F. Hughes. Sketch: An interface for sketching 3d scenes. In *Computer Graphics Proceedings, Annual Conference Series*, pages 163–170, 1996.