1 Introduction

In the introduction of a paper concerning artistic rendering, it is typical to note that the early development of computer graphics was mainly concerned with photorealism. This is true, of course, but only when we constrain our view to *computer* graphics and exclude computer-aided design. The larger field of graphics has long concerned itself with other criteria for imagery: aesthetics, information content, etc. It was probably inevitable, then, that these criteria became important considerations for researchers in computer graphics. Up to this point research in non-photorealistic rendering seems to have been focused on the duplication of traditional art forms such as watercolor, pointillism, and mosaic. Part of that trend, the purpose of this project has been to experiment with the generation of pen-and-ink style illustrations in an automated fashion from photographs.

In the opinion of this author it is unnecessary to provide justification for work in non-photorealistic rendering that has any sort of aesthetic value. For those that disagree, however, justification can be taken from the uses of pen-and-ink illustration in textbooks. Winkenbach and Salesin [9] state that pen-and-ink illustrations are used in manuals and medical texts because they abstract away unnecessary detail that may distract the viewer. A more pragmatic concern is that pen-and-ink illustrations can be printed without the use of halftoning algorithms or expensive printers. Whatever the justification, though, the automatic generation of these illustrations is an interesting intellectual task.

For the purposes of this project we will consider the generation of pen-and-ink illustrations as a two-part process. The first part involves the generation of shape outlines from the photograph while the second involves shading regions of the illustration to convey both the tone and the texture of the input picture. It should be noted that outlines are not necessary in pen-and-ink illustrations, but the identification of object boundaries is necessary in order to perform shading in a sensible way.

2 Objectives

The main objective of this paper was to devise and implement an automatic method for generating pen-and-ink illustrations from photographs. Moreover, having separate modules for the generation of outlines and area shadings will allow for experimentation with related illustration styles. Having generated outlines in the pen-and-ink style, for example, it would be straightforward to color image regions in such a way as to give a cartoonish appearance.

3 Previous Work

The computer-generation of pen-and-ink illustrations seems until recently to have been the work of one group at the University of Washington. The key points of some pen-and-ink papers are described in the following sections.

3.1 Winkenbach and Salesin ([9])

In this paper the authors describe an automated method for generating pen-and-ink illustrations of architectural models. This was motivated in large part by the documentation of well-established

conventions for their representation which give direction to their work.

One of the key points of this paper is that each stroke conveys both shading and texture of the underlying region. Because of this, the medium is inherently limited - the need to convey a region's texture limits the range of luminance that can be reproduced there. Because of this, and the fact that strokes are generally larger then individual pixels, tones are only reproduced in a rough sense. While individual tones may not be accurately portrayed, it is necessary to portray the tones accurately in a *relative* sense - dark regions in the scene should correspond to dark regions in the illustration.

The authors make the interesting note that outlines, which don't exist explicitly in realistic scenes, are the primary means of conveying form in children's illustrations. Perhaps this is a comment on the efficiency with which they convey information, or perhaps it is a comment on the level of sophistication in children's illustrations. Because pen-and-ink illustrations are composed of strokes that are generally not connected, *boundary outlines* are necessary to indicate where regions intersect. Artists also use *interior outlines* to give the impression of shape or depth by highlighting discontinuities in brightness.

On a slightly lower level, the authors give some general rules for the types of strokes that should be used for different materials. In particular, they mention that

- Glass is best represented by crisp, straight lines.
- Sketchy lines are good for old materials.
- Absence of detail indicates the presence of glare on a shiny surface.

Each stroke generated by this method is composed of several functions: the *path* P, *nib* $\mathcal{N}$, and *character*. The path is simply a parameterized curve that maps real values in the unit interval to points in the plane through which the stroke will pass. The nib represents the brush used to paint the stroke, and expresses the width of the stroke as a function of the pressure applied. The character function, which is also parameterized on the unit interval, gives the waviness C_w and pressure C_p of the stroke at each point.

Rendering a stroke generates a list of pixels to color based on the formula

$$\mathcal{S} = (P(u) + C_w(u)) * \mathcal{N}(C_p(u))$$

where

$$A(u) * B(u) = \bigcup_{u \in [0,1]} \{a + b | a \in A(u) \; and \; b \in B(u)\}$$

In summery, this paper provides good detail describing how each stroke should be rendered. Furthermore, it summarizes some of the conventions of the medium that should be enforced when generating illustrations. It does not, however, provide much detail in the area of stroke generation. We might supposed that the architectural model has some information about the type of the material, making this problem less difficult.

3.2 Salisbury, et al [6]

This paper, which was published concurrently with the one described in the previous section, describes an interactive method for generating pen-and-ink illustrations from reference images. The

method is intended to reduce the amount of work needed to produce an image, and does so by semi-automatically generating pen strokes based on the user's command.

A large part of the paper is spent discussing the stroke textures that are available to the user. Each texture is a collection of strokes that have a consistent look, for example cross-hatching. Moreover, the strokes that comprise each texture are prioritized by their importance in creating the sense of texture. This gets back to the point that tone and texture are linked in pen-and-ink illustrations, as the highest-priority strokes are used to shade lighter areas of the scene while the lower-priority strokes are reserved for darker areas.

The authors also discuss the interplay between outline and shading strokes in illustrations. In particular, their application forces shading strokes to be clipped at outline strokes. Moreover, they suggest that the point of clipping should include some randomness so that strokes don't all end at the same location relative to the outline. This has the effect of softening the edges, as well as giving the illustration a hand-made look.

The reference image is used for several purposes in this application. To the user the image serves as a visual reference in whatever ways she may desire. To the application, the image's tone serves as a reference for the brightness while the gradient serves as a guide for stroke orientation. Furthermore, the edges in the reference image are extracted in order to generate the outlines, though the authors do not describe their method for doing so.

As with the previous paper, this one outlines a number of rules for strokes without actually saying how they are generated. The user who interacts with the application is the primary force in stroke creation, so this paper is little help when attempting automatic illustration.

3.3 Winkenback and Salesin [10]

The main contribution of this paper is the generation of illustrations based on a 3D model of a surface which has been parameterized. This method involves both the 3D model and a set of user-specified parameters such as the maximum stroke thickness.

The authors introduce a method called controlled-density hatching by which a sense of shape can best be conveyed in brighter areas of the scene. Depending on the parameterization of the surface, it may be the case that following contours of one of the parameters leads to higher stroke density - and thus a darker tone - in certain regions of the surface. By applying the controlled-density hatching method, strokes are truncated to portray the correct tone while still conveying the shape of the underlying surface.

While the results of this paper are quite impressive, the method is not immediately applicable to automatic image-based illustration because there are no parameterized surfaces to render. Indeed, the generation of such surfaces would no doubt involve some rather complicated computer vision techniques.

3.4 Salisbury, et al [7]

In a refinement of their previous work, this paper focuses on an interactive system for generating pen-and-ink illustrations from reference images. In this version the application uses three inputs to generate an illustration. The first is the image, which is used as a reference for both tone and edges. Secondly, the user generates a vector field that specifies the direction of strokes for each point in the image. Finally, a set of example strokes are provided which express the notions formerly given by the character function.

In addition to describing the tools with which the user generates these inputs, the paper gives a method for generating strokes. At the heart of this method is what they call the importance image. This image is essentially the difference between a blurred version of the illustration and the reference image. The idea is that, starting with a blank canvas, we iterate to drive the importance image to zero. That is, we add strokes in areas of the illustration where the tone differs significantly from the input image and continue until the residual is close to zero.

While the interactive portions of this application are not relevant to our goal of automatic illustration, the iterative algorithm provides good guidance for shading of image regions.

3.5 Deussen and Strothotte [1]

This paper provides a detailed treatment of a very small problem: the pen-and-ink illustration of trees. The system that is described, like the previous papers, depends on a 3D input which represents the tree. Depth values in this model are used as the basis for generating object outlines that will be the main feature of the illustration.

The authors of this paper make the decision to render the tree's trunk and leaves separately. The trunk, in turn, is modeled as a set of generalized cylinders that represent the root and branches up to the second order. These cylinders are shaded using the iterative technique described earlier.

For the leaves, the authors introduce a parameterized approach that allows a user to determine the amount of detail shown. The two parameters are the size of each leaf and a depth threshold. The depth threshold controls the number of outline strokes drawn by eliminating those edges that do not indicate a large enough difference in depth, thus allowing for an abstract representation of foliage.

While this method is interesting, it doesn't give any meaningful insight to the problem of automatically rendering illustrations from images. Detecting trees would be a difficult vision problem, but determining the parameters of this rendering would force more choices by a user.

4 Where Does This Leave Us?

Having reviewed the literature, it seems that automatic illustration from images may not be so straightforward. Whereas all of the methods use 3D models and/or user interaction to some extent, this method must derive the same information from the image alone. In particular we must generate outlines from edges in the image and attempt to generate the path, nib, and character functions automatically.

Having done this, shading can be accomplished using the iterative method described by Salisbury, et al. Without a user to direct the strokes, however, it is necessary to generate the direction vector field from image cues.

The following two sections describe the work done on the outline and shading portions, respectively. As the purpose of this project was to try different methods, each will be described as a series of experiments.

5 Generating Outlines via Edge Detection

Edge detection has long been a goal of computer vision. While hundreds of algorithms have been presented, it may still be considered an open problem because there is no commonly accepted

definition of what constitutes an edge. The term is horribly overloaded, with each application of edge detection having a different - often contradictory - definition in mind.

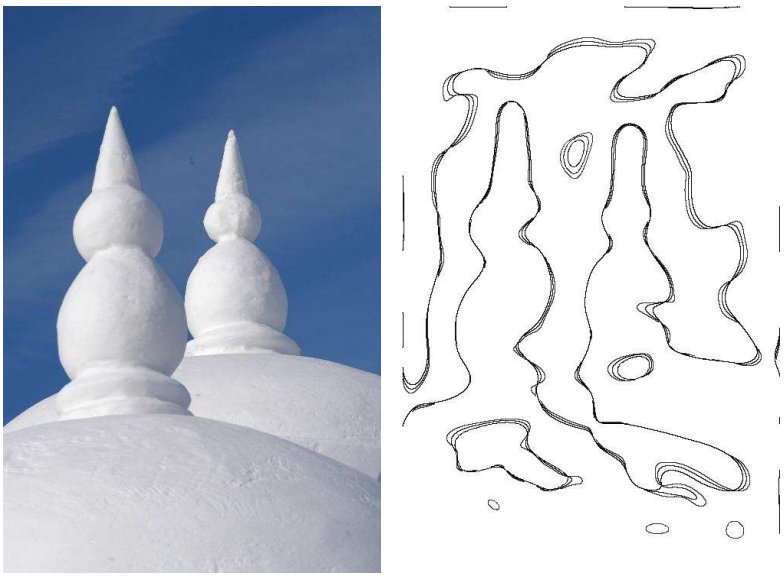
5.1 Edge Detection a la Marr and Hildreth

One of the classical methods of edge detection was introduced by Marr and Hildreth [5], and is based on a model of human vision. Physiological experiments with cats proved the existence of cells in the visual cortex that respond in the presence of edges in some fixed orientation. These so-called center-surround cells detect oval-shaped bright regions surrounded by dark regions (or vice versa).

Marr and Hildreth's edge detection scheme works by simulating these cells by filtering the image with a kernel whose center coefficients are negative and whose outer coefficients are positive. This kernel is actually the Laplacian of a Gaussian, and edges are said to be present where the filter's response crosses zero. This method is typically parameterized by the variance of the Gaussian, which determines the size of the kernel and thus the extent of the window in which we look for edges.

The Marr and Hildreth operation is performed on grayscale images, so it is necessary to transform our given (RGB) image to monochrome. This is done in the normal manner by taking a linear combination of the red, green, and blue values at each location.

This method proved to be troublesome for several reasons. First of all, edges detected using a small scale (variance) included many in areas of slight changes in shading. Using a larger scale gave the intended number of edges, but their locations were effected by the extent of the filter. See figure 1, which shows the *snow* image and the result of the Marr Hildreth edge detection with a high variance. Another problem has to do with the fact that edges are found in the grayscale image, ignoring transitions between differently colored regions of approximately the same brightness. For these reasons, it was decided to try something new.



(a) Snow Image (b) Marr Hildreth Edges

Figure 1: A First Stab at Edge Detection

5.2 Taking a Step Back

Before attempting some other edge detection method, it was important to decide what we're really looking for. Because the shading methods would later include some sort of color averaging, it makes sense to look for edges in the image that separate areas of significantly different color. Having said this, however, it is worth noting that we have defined one term with terms that are, themselves, undefined. What is *significantly different color*?

The naive answer would be to find the Euclidean distance between two points in the image's native RGB color space. If this distance exceeds some arbitrary threshold, we might consider the two colors to be significantly different. While the simplicity of this method is tempting, it depends on the underlying RGB color space. This is unfortunate because, while RGB is typically used for convenience, it has a number of undesirable characteristic.

One such characteristic is that RGB space is *perceptually nonuniform*, meaning that distances between colors are essentially meaningless. While the distance between the points [255, 245, 235] and [200, 190, 180] is the same as the distance between the points [155, 165, 175] and [100, 110, 120], the magnitude of the perceived difference between the colors with these RGB coordinates is not the same. If we are going to generate edges by looking for large changes in color, then, we need a different metric.

One metric that has been designed for just this task is CIE Δe^* [3]. Without going into great detail, the Δe^* between two colors is the Euclidean distance between their coordinates in a color space called CIE$L^*a^*b^*$. CIE$L^*a^*b^*$ is a luminance-chrominance color space which is obtained by a non-linear transformation from RGB.

5.3 Edge Detection from Δe^*

Given an RGB image and a threshold Δe^* value, our revised edge detection is performed by the following steps:

1. Convert the RGB image into an $L^*a^*b^*$ image.
2. For each pixel, determine the Δe^* between it and each of its 8 neighbors (horizontal, vertical, and diagonal). If any of these 8 values exceeds our threshold, call the current pixel an edge pixel.
3. Group adjacent edge pixels into edge contours.

A result of this edge detection method is shown in figure 2. While the edge pixels are now in the expected positions, the boundaries between objects aren't necessarily complete. This clearly presents a problem when we attempt to shade a region with the average color of the pixels therein.

Based on this requirement, we really need a segmentation method. Given a segmentation of the image, edge pixels can be identified as those who have at least one neighbor belonging to a different segment. More importantly, membership in a region is clearly specified by segmentation, which is not always the case with edge detection.

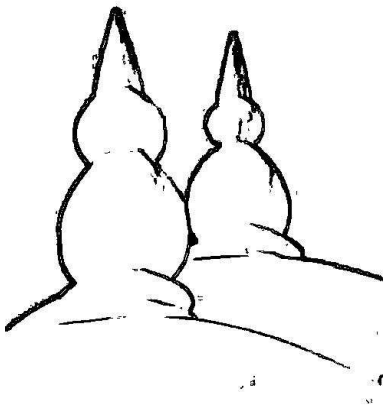


Figure 2: Edges Found using Δe^*

5.4 Segmentation using Δe^*

The general approached used for this segmentation was to look at a pixel's neighbors and the Δe^* values between them. If none of those Δe^* values are below a given threshold, we claim to have discovered a pixel in an entirely new segment. Otherwise, we consider this pixel to be a member of the segment that includes its closest (color sense) neighbor. This tends to generate a large number of segments with fairly similar color due to spatial separation, so the initial segmentation is revisited, merging those regions that have a similar average color. Finally we enforce a size constraint, merging segments with too few pixels into the closest (color sense) region. Because all of this merging has likely changed the average color of the segments, the final segmentation is based on each pixel's Δe^* with respect to the average color in each segment.

Each of these steps is described in greater detail in the following subsections.

5.4.1 Initial Segmentation

We visit the pixels in raster-scan order, starting with the second pixel in the first row. For each pixel we calculate the Δe^* value between its color and that of the (as many as) four neighbors that we have previously visited. If none of these values are within our threshold, we say that the pixel belongs to an entirely new segment. If at least on of these values is below the threshold we say that the current pixel belongs to the same segment as the neighbor with the lowest Δe^*.

Because of the fact that each pixel is only compared to those that have already been visited in a raster order, we tend to get segments whose major axes go from upper left to lower right. This tendency results in our missing edges that are generally in the same direction. In order to address this, the first segmentation is performed on the image in its given orientation and rotated by 180^o. The segments output from this step consist of pixels which are grouped together in both of the initial segmentations.

5.4.2 Merging Segments

In order to merge segments, we consider each in order of its segment number. We compute the average $L^*a^*b^*$ value of all of the pixels, and compare it to the same value for each other segment. If the smallest Δe^* is below the same threshold that we used in the initial segmentation, the current segment is merged with the one corresponding to the minimum Δe^*.

5.4.3 Enforcing a Minimum Segment Size

We count the number of pixels belonging to each segment and determine the percentage of the original image that it represents. If that percentage is less then some threshold, we merge the region with the segment with the most similar color.

5.4.4 The Final Pass

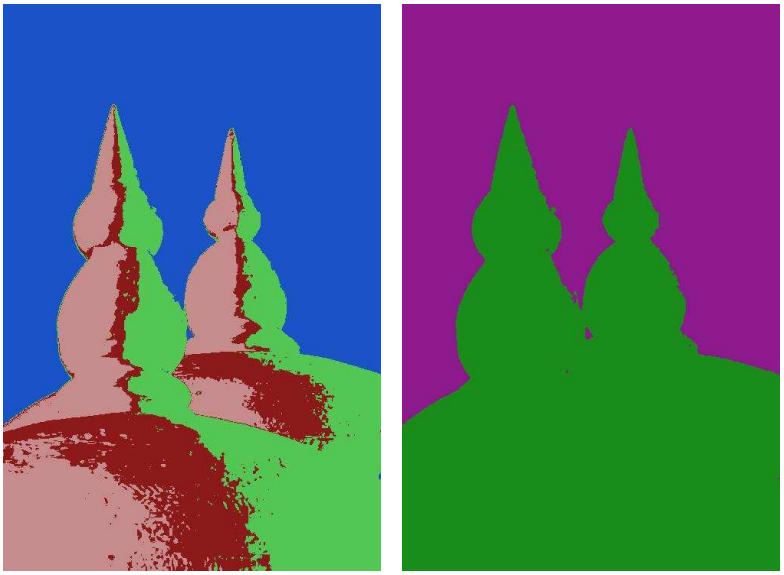
As a result of merging different segments, we may now have pixels whose color is closer to the average of a different region. In order to minimize this, we compute the Δe^* value between each pixel and the average color of every region. Each pixel is assigned to the segment with the lowest Δe^*. We assume that the number of pixels that change labels in this step is relatively small, and that their change has a negligible effect on a segment's overall average.

5.5 Segmentation Results and Timing

Figure 3a shows the result of segmenting the snow image. The parameters of the segmentation were $\Delta e^* = 13$, and $thresholdSize = 2$. The segmentation is slow - approximately 100 seconds for an image of 742-by-493 pixels - in part because it's implemented in Matlab code.

The segmentation has clearly picked out the sky to be a unique region, and has segmented the show into three regions based on the surface's orientation with respect to the sun.

Figure 3b shows the segmentation resulting from a higher threshold: $\Delta e^* = 15$. It now considers the snow to all be one region, but some of the darkest areas of the snow, which show a blueish hue, are grouped with the sky region.



(a) Threshold $\Delta e^* = 13$ (b) Threshold $\Delta e^* = 15$

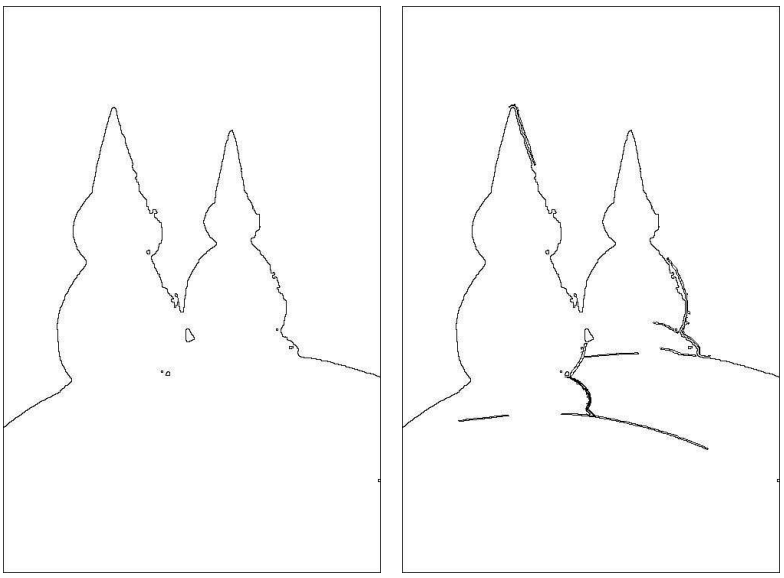
Figure 3: Segmentation of the Snow Image

5.6 Drawing Outlines from a Segmentation

Given a segmentation, outlines can be found by looking for pixels whose neighbors have a different label. While this defines a set of outline points, we would prefer a set of outline contours. In order to generate contours from the outline points, we use the square tracing algorithm as described in [8].

A binary image is created to look like a pen-and-ink style outline by darkening each point on each of the contours. All other points are left white. An example is shown in figure 4a.

In addition to drawing outlines at segment boundaries, the previous work on pen-and-ink suggests the use of lines to convey shading discontinuities. This has been implemented by looking for changes in the image's luminance, and generating contours of points where that change exceeds some threshold. Figure 4b shows the outline from figure 4a with these edges added.



(a) Outlines Only (b) With Luminance Edges

Figure 4: Outlines of the Snow Image

6 Shading Segments

Now that the image has been segmented and the outline has been drawn, we can consider coloring each region. Each of the shading methods described below plugs in to a general framework. The framework breaks the image into blocks and shades each individually. If the block's pixels belong to more then one segment they are masked so that the shading method only handles pixels in one group at a time. In this way we can tile the image while respecting segment boundaries, though the tiling effect can be avoided by setting the block size to equal that of the image.

6.1 Pen-and-Ink Style Shading

Within each block, we first define the set of pixels that we can darken to be those that are further then three pixels from the nearest point on the outline. Then we generate our original shading - a white canvas of the same size and shape as the block.

We then run the iterative darkening method described previously. While the average grayscale value in the shading is less then the same value in the original image, we perform the following steps:

1. Find the pixel location where the difference between the current and reference shading is largest. Call that pixel the starting point of our stroke.

2. Look at the neighboring pixel in the direction perpendicular to the gradient. If that pixel is valid (not too close to an outline pixel), call it the next pixel in the stroke and repeat this step for it.

3. Update the map of valid pixels so that future strokes do not intersect this one.

4. Draw the stroke by darkening each of its points.

Note that the update to the map of valid pixels is necessary in order to avoid dark regions in the reference image becoming blobs of ink in the illustration.

The results of this method were rather disappointing. An example is shown in figure 5, and we notice a few effects. First, while there are light and dark parts of the image, the contrast is too high. Because of the requirement that strokes not intersect one another there is effectively a non-zero minimum brightness that can be reproduced. Moreover, that minimum seems to be fairly high, so there are a lot of image regions that have the same tone.

Secondly, the strokes are short and, relative to the size of the image, too narrow. The shortness has to do with the fact that the field of vectors normal to the gradient was noisy, implying that the image should be blurred ahead of time. The small width has to do with the brush size, which was chosen to be 1.

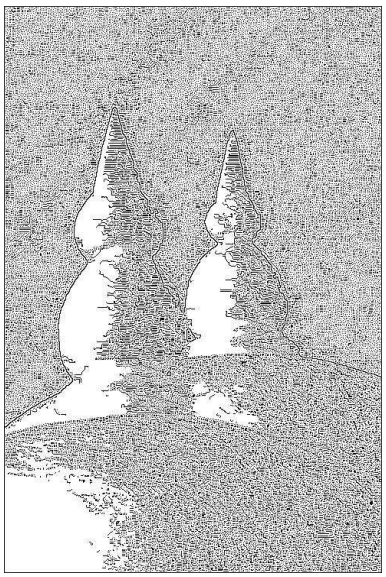


Figure 5: Pen and Ink Illustration

6.2 Average Coloring and Toon Shading

Having reviewed a tutorial on toon shading ([2]), it seems that a high-quality segmentation would allow one to generate a cartoon-style illustration from an image. According to the tutorial, each segment should be filled with solid, unlit color. Leaving aside the fact that color is undefined in the absence of light, this would seem to imply that we could fill each region with its average color and get something that looks like a cell of animation.

There are a couple of variations that may be interesting depending on the source image. First, we leave it as an option whether or not to display the outline in the final image. Second, since the lack of texture in the toonshaded image is sometime boring, we introduce a blending parameter in the range [0,1]. A value of 1 means that the average is used at each location. A value of 0 means that the texture of the original image (derived from the luminance channel of the $L^*a^*b^*$ separation) is used with the average chrominance. Values between 0 and 1 are a weighted combination of the two.

The function that performs this shading is simple and runs relatively fast despite the fact that it's written in Matlab. The parameters are the RGB image, a drawing of the outlines, a map of its segmentation, and the blending and outline parameters just described.

Figure 6 shows a toon shading of the snow image for the two extrema of the blending parameter. Neither show the outline, as it's somewhat distracting.

7 Some Results

Results are shown at the end of the paper for two different images.

8 Future Work

There are certainly significant improvements that could be made to this project if time allowed. Here are a few:

(a) Blend = 0 (b) Blend = 1

Figure 6: Toon Shading with Different Blending Parameters

1. Other then the choice of metric, the segmentation method was rather heuristic. A survey of color segmentation literature may reveal a method that is better, faster, or both.

2. The current segmentation algorithm should be changed in order to account for spatial locality. Often the segmentation will result in small islands of one segment in the middle of an ocean of pixels from another. While this may be valid in some cases, generally the pixels are not sufficiently different from their neighbors to justify the creation of another outline.

3. Merging segments should probably be done in order of the segment's size. This would cause the average color of a segment to converge in a more predictable fashion, and possibly reduce the number of pixels that change labeling in the final step.

4. Given a segmentation, it would be useful to generate a parameterized representation of the boundaries. This would allow us to come up with the character function (from the curvature and gradient magnitude), giving the ability to create stylized outlines. It would also allow for the enforcement of a smoothness criteria.

5. The pen-and-ink style shading didn't work well, in part, because the gradient field was rather noisy. There are a number of approaches that might help here: blurring the original image or doing gradient interpolation are two choices that come to mind.

6. Generating image-based mosaics that respect segment boundaries. This would fit nicely in the shading framework developed. Indeed, code to create such mosaics was written, but the data collected on the set of image blocks was faulty, giving a terrible result.

7. Based on the hue of the sky in the toonshading example, we might be inclined to use something other then the average chrominance for a region's color. Depending on how the a^* and b^*

coordinates are distributed, it may make more sense to use the centroid of the region, rather then the mean in both directions.

9 Conclusion

This project was largely inconclusive due to a lack of time. The most interesting parts would probably have been the stylization of outlines. Unfortunately, the effort involved in simply generating them was significant in large part because the starting point (Marr-Hildreth edge detection) was so far from the intended method. It still seems like a feasible goal, however.

Shading in the style of pen-and-ink seems to be a more difficult task to perform automatically. Given the amount of extra information used by the previous applications in pen-and-ink, it seems unlikely that an interesting shading can be produced with a reasonably small and straightforward set of parameters. It may very well be possible to produce a shading that follows many of the conventions, which would be useful for textbook-style illustrations. Generating *interesting* pen-and-ink illustrations, however, might prove to be more difficult.

At the risk of editorializing, this seems to be a problem with many non-photorealistic rendering methods. We may be able to generate some consistent set of results, but the flexibility necessary to create *interesting* examples seems inevitably to come at the expense of usability.

Like early artistic photographers who smeared Vaseline on their lenses, computer vision researchers seem to be focused on the duplication of traditional artistic techniques. While this is interesting and, to a certain extent, liberating one gets the impression that there's something genuinely new to be done with computer graphics. The advantages of photography over painting eventually gave rise to documentary photography, and it seems inevitable that the advantages of computer graphics will be exploited in some new and exciting way.

References

[1] O. Deussen and T Strothotte *Computer-Generated Pen-and-Ink Illustration of Trees.* Proc. SIGGRAPH 2000.

[2] C. Graf. *A Tutorial on How to Create Cartoon Shaded Figures from 3D Models.* Course Project. http://www.cs.uni-magdeburg.de/~cgraf/NZ/COMPSCI715/Project/Tutorial.html

[3] R. W. G. Hunt *The Reproduction of Colour* England: Fountain Press. 1995.

[4] J. Lansdown and S. Schofield *Expressive Rendering: A Review of Nonphotorealistic Techniques.* IEEE Computer Graphics and Applications, 15(3):29-37, May 1995.

[5] D. Marr and E. Hildreth *Theory of Edge Detection.* Proceedings of the Royal Society of London. Series B, Biologival Sciences, Volume 207, Issue 1167, p 187-217. February 29, 1980.

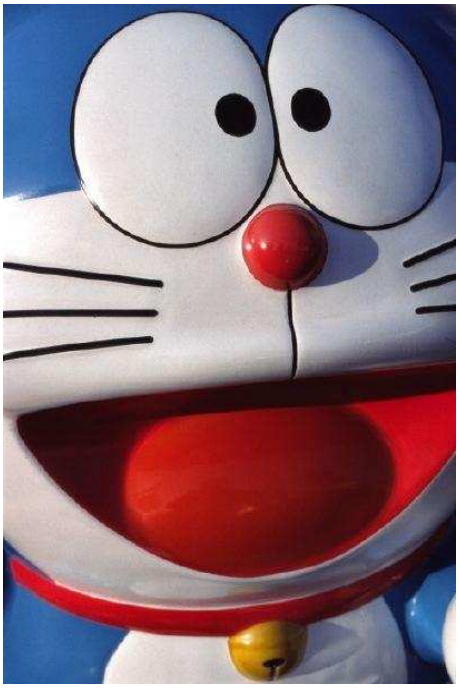
[6] M. Salisbury, S. Anderson, R. Barzel, and D. Salesin. *Interactive Pen and Ink Illustration.* Proc. SIGGRAPH 1994. Orlando Florida. 1994.

[7] M. Salisbury, M. Wong, J. Hughes, and D. Salesin. *Orientable Textures for Image-Based Pen-and-Ink Illustration.* Proc. SIGGRAPH 1997. Los Angeles, California. 1997.

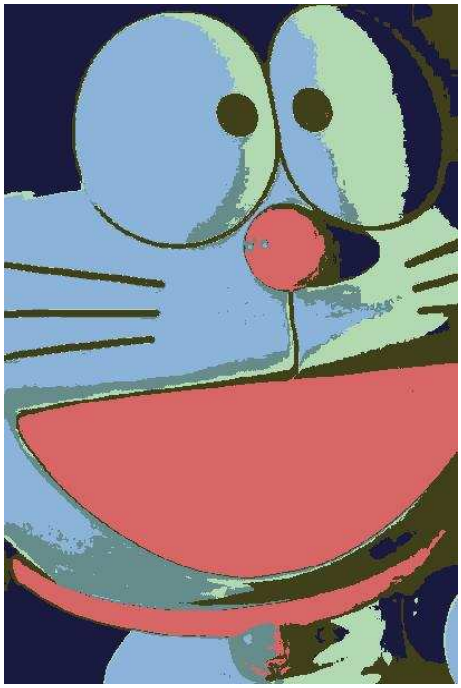
[8] G. Toussaint *Grids, Connectivity, and Contour Tracing.* Lecture notes. http://www-cgrl.cs.mcgill.ca/~godfried/teaching/pr-notes/contour.ps

[9] G. Winkenback and D. Salesin *Computer-Generated Pen-and-Ink Illustration.* Proc. SIGGRAPH 1994. Orlando, Florida. 1994.

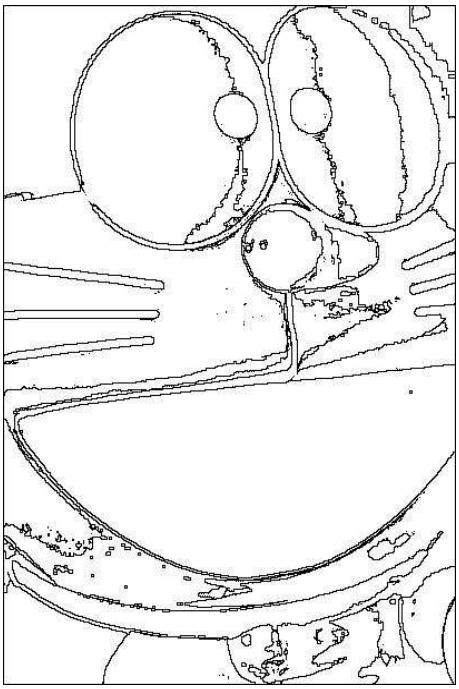
[10] G. Winkenback and D. Salesin *Rendering Parametric Surfaces in Pen and Ink.* Proc. SIGGRAPH 1996. New Orleans, Louisiana. 1996.



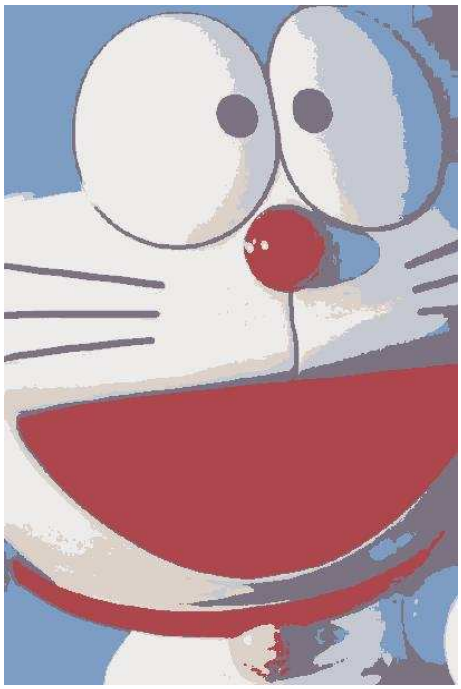
(a) Original Image



(b) Segmentation - Threshold $\Delta e^* = 27$, size = 5



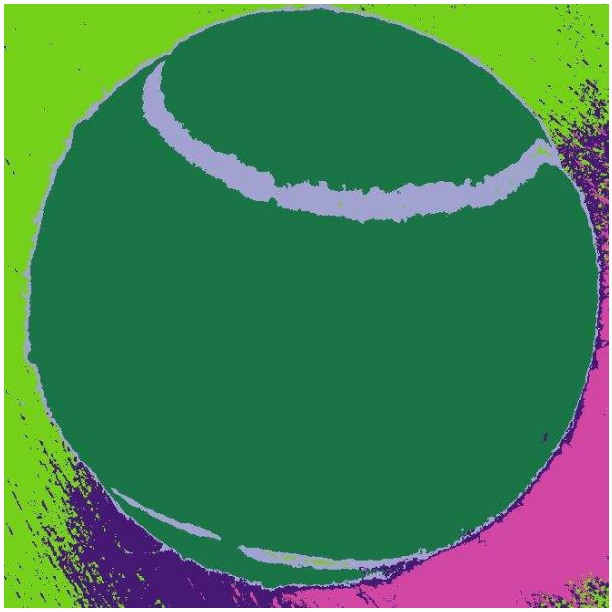
(c) Outline



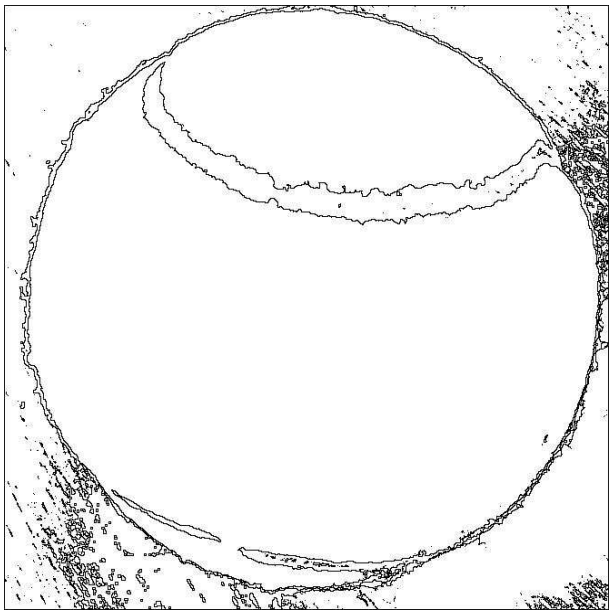
(d) Toon-Shaded, Blend = 1, No Outlines

Figure 7: The Cartoon Guy Image

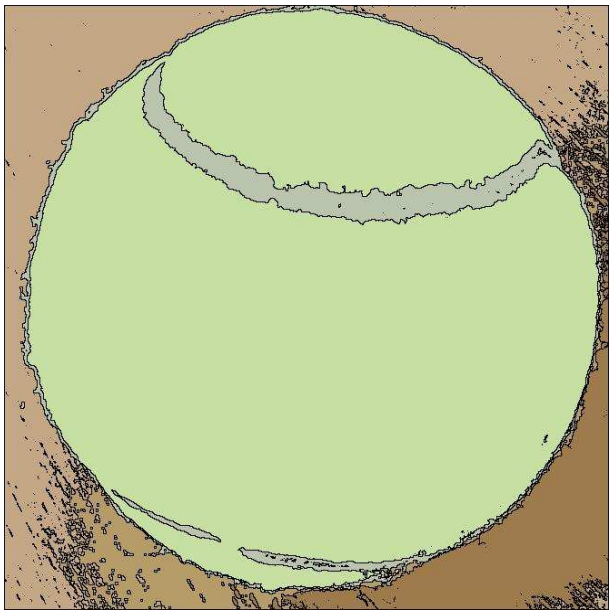
(a) Original Image



(b) Segmentation - Threshold $\Delta e^* = 10$, size = 2



(c) Outline



(d) Toon-Shaded, Blend = 1 with Outlines

Figure 8: The Tennis Ball Image