CEL SHADING IMAGE PROCESSING PROJECT

Sharud Agarwal, Helen Kuo

ABSTRACT

Cel shaded images are often regarded as "cartoon" looking images and in fact are often called toon shaded images. It is a very distinct artistic style that is used for some television shows, movies, and games; however, it is not used often in still camera images. We wanted to find out why there aren't many popular cel shading programs available. We did this by developing an Android application that can take images captured using the device's camera and post-processing them to make the images cel shaded. Although we met with some success, there were some images that would not turn out well.

1. INTRODUCTION

In 2012, 82% of people used their mobile phone to take pictures and that number continues to grow [1]. This, along with the rise of image filter apps available show how much users value taking pictures and customizing them. One very popular image style is cel shading. It is an artistic style where the media looks "cartoony" and it is very popular in multiple forms of entertainment media. However, it surprisingly is not very popular in mobile media applications for mobile users to use. We wanted to create an Android application that post-processes images taken on the phone camera and makes them cel shaded.

Mobile development has its own requirements that we believe have been keeping this art style away from phones; the app must be fast, easy to use, and produce an image that is appealing to the user as often as possible. For cel shading to be accurate, however, it usually uses costly algorithms and for it to be fast, the algorithms often sacrifice accuracy. With our approach, we looked to find a middle ground that produces accurate cel shading while also not being so computationally expensive that it is too slow for users.

1.1. OpenCV4Android

OpenCV4Android is an Android SDK for the OpenCV library. Because it is meant specifically for Android, some functions are optimized for mobile [6]. Because it is optimized for mobile, we are able to use it to do some computationally expensive image processing necessary for cel shading relatively fast on a mobile device.

1.2. Problem Statement

In order to create a cel shaded image from an already captured still image, there are several post-processing steps one must take. The first step is to detect and extract the edges [2]. The edges must not be too bold so to avoid looking more like blobs than edge lines, but also not too thin such that they are not very pronounced. The edge map is a very important element to cel shading and thus was the focus of our testing. The next step is to smooth the regions on the images to remove high frequency details that are not important to the image. This step makes the following quantization step work better. Then this new image is quantized to remove some of the gradients in the colors. This essentially lowers the number of color shades necessary to represent the image by reducing the bits per pixel. This helps to create the "cartoony" effect of the image. The final step is to overlay the edge map extracted earlier and place it over the newly quantized image.

1.3. Related Work

The Flow-Based Image Abstraction paper by Henry Kang, Seungyong Lee, and Charles K. Chui [2] is the most successful implementation of a cel shading algorithm we found. However, it focuses on flow-based algorithms which are much more computationally expensive than the original algorithms. We attempted to use the research shown in this paper and choose the best route for a mobile application.

2. EDGE DETECTION

Edge detection is an essential step to cel shading. Too many edges can result in an image that becomes too cluttered to find the important details and too few edges results in images without enough details. For this reason we tested many different edge detection algorithms.

2.1. Gradient Edge Detection

Gradient edge detection algorithms are the first edge detection methods we tried. The gradient of a 2D image is a two element vector of the partial derivatives of the image. Gradient edge detection works well because gradients are "rotationally symmetric or isotropic" [3]. This allows for edge detection regardless of orientation. This method works through three main steps: discrete differentiation, point operation, and threshold. The first two steps are used to enhance the edges while the threshold step is used to detect the edges. Discrete (or digital) differentiation digitally approximates the gradient by differencing the image.

Differencing, however, is highly sensitive to noise as it emphasizes high frequencies. For this reason, we used two different 3x3 edge templates (kernels) that are designed to reduce noise to convolve the image with to produce the directional derivative estimates. The first 3x3 kernel we used was the Sobel operator which uses kernels [3]:

Δ_x = [-1 0 1; -2 0 2; -1 0 1]

Δ_y = [-1 -2 -1; 0 0 0; 1 2 1]

These are able to denoise by averaging the rows and columns respectively. The Sobel operator, however, does not maintain perfect rotational symmetry. The Scharr operator, as stated in the OpenCV API, looks to fix this with kernels:

Δ_x = [3 0 -3; 10 0 -10; 3 0 -3]

Δ_y = [3 10 3; 0 0 0; -3 -10 -3]

The next step was the point operation, which simply produces the gradient magnitude from the directional derivatives obtained in the previous step. This is done by squaring each directional derivative, summing the two squares, and taking the square root of the sum. Finally, a binary edge map is produced by thresholding the gradient magnitude.

Gradient edge detector algorithms are simple and cheap, thus lending themselves well for mobile applications. However, they are noise sensitive and do require a threshold to be selected. Because the app needs to be easy to use and fast, the threshold will need to be preselected in the code so the user does not have to choose a threshold. We will preselect the threshold by equalizing the image and then using the mean of the pixel values as the threshold [5]. Also, although often thought of as a negative, gradient edge detection produces thick edges (few pixels wide as opposed to one or two pixels wide), which can be good for cel shading. The results for these filters can be seen in the results section.

2.2. Difference of Gaussians

Another method of edge detection is to use a difference of Gaussians (DoG) filter. A Gaussian filter is a low pass filter and thus the DoG filter is a band pass filter. A DoG filter takes two Gaussian filters, one of which is less smoothed (or blurred) than the other, and takes the difference of the two. Mathematically, a Gaussian filter is a convolution of the image with a Gaussian function. The 2D function used for this is [4]:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The distribution of this is shown in **Figure 1** below.

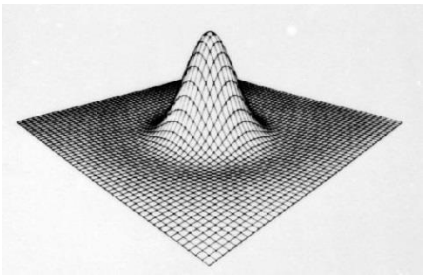

Figure 1

2.3. Canny Edge Detection

The final edge detection method we tested was the Canny edge detection algorithm. This algorithm was designed to improve upon Laplacian-of-Gaussian filters [3]. The steps to the Canny algorithm are to smooth the image using a Gaussian filter as in section 2.2. Then compute the gradient magnitude and orientation as in section 2.1. After that, the second derivative of the smoothed image is computed and the zero-crossings will be taken from the image. These zero-crossings form the edge map. The Canny algorithm combines both of the previous methods and thus we expect it to be the best performing of the three methods we tried.

3. REGION SMOOTHING AND QUANTIZATION

The next step in cel shading is to create the cartoon color effect. This is done by quantizing the colors and smoothing the regions such that small unimportant details are removed [2]. First step is to smooth the regions then we will quantize the colors.

3.1. Bilateral Filtering

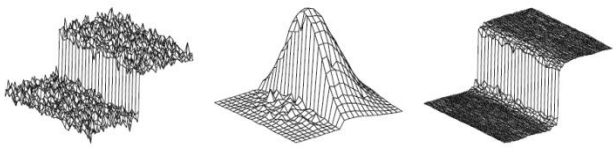

Figure 2

Bilateral filtering is an image smoother than looks to maintain edges [3]. It is a non-linear filter because it combines spatial distance weighting with luminance similarity weighting. **Figure 2** above is an example of a noisy edge that is bilateral filtered:

The image on the left in **Figure 2** is the noisy edge of an image, the middle image is the bilateral weighting of a pixel near the edge, and the image on the right is the denoised (smoothed) edge. We did notice that the bilateral filter only made a small difference on images from the Android camera and thus, as discussed later in the results, we tried doing an iterative bilateral filter as was done in the Flow-Based Image Abstraction paper [2]. Bilateral filtering is computationally expensive however [3], so the number of iterations has to be limited, especially with larger window sizes.

3.2. Gaussian Smoothing

Gaussian smoothing is another method of region smoothing we considered. Although it is not as computationally expensive as bilateral filtering, it does not maintain edges as well because it does not do both spatial and luminance weighting.

3.3. Color Quantization

Color quantization is done to reduce the gradients in the colors and make the colors look more like blobs of the same color. This is an effective technique to creating cartoon-like images. Color quantization is done similarly to grayscale quantization. Each channel is quantized by doing integer division on each pixel and then multiplying the result by the same divisor as shown in the equation below:

$$J(i,j) = \left\lfloor \frac{I(i,j)}{x} \right\rfloor * x; for\ all\ i,j\ in\ I$$

This is the final step to creating a cel shaded image before combining the edge map and the filtered image.

4. COMBINING EDGE MAP WITH IMAGE

The final step is to overlay the edge map onto the newly created image after region smoothing and quantization. This is done by taking the pixel locations that are black on the edge map and making those same pixel locations on the filtered image black. This results in the final cel shaded image that gives images taken on the camera a cartoony art style.

5. RESULTS

5.1 Edge Detection

Our first step was to produce an edge map from an image taken from the camera in Android. We started by using the Canny algorithm for edge detection and automatically selected a threshold using the Otsu algorithm. We came to realize though that the Otsu algorithm is more suited for images with distinct edges. We found a new method of selecting a threshold where we equalize the image and use the mean as the threshold [5]. However, as shown in **Figure 3** below, the mean threshold did not produce very good edges as opposed to a lower threshold of 80 and an upper threshold of 90 that was suggested by the OpenCV API.

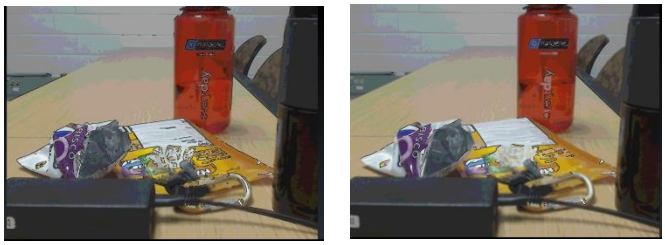

Figure 3 a) 80/90 thresholds b) mean threshold

Thus we went with the 80/90 threshold for the default thresholds in the Canny algorithm.

After this we tested the DoG edge detector and the Sobel and Scharr gradient edge detectors. As shown in the figures below, the Canny and DoG algorithms worked the best, with Canny producing thin edges throughout and DoG producing thicker edges. The DoG filters have sigma parameters that affect the edge detection significantly. As we learned in Homework 2, problem number 3, a sigma1 value of 3 or 4 worked best. The sigma2 value was equal to 1.6 times the sigma1 value [3].

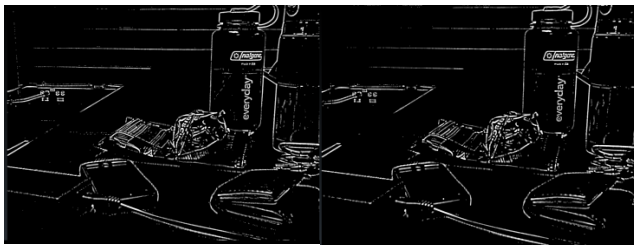

Figure 4 DoG a) Sigma1 = 3 b) Sigma1 = 4

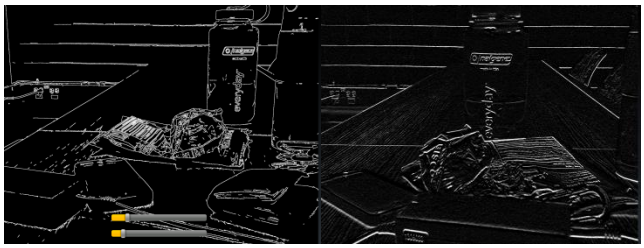

Figure 5 a) Canny b) Scharr in y-direction

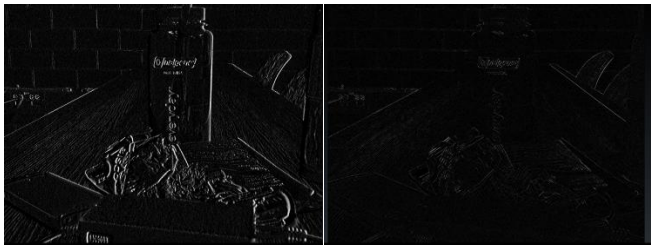

Figure 6 a) Scharr in x-direction b) Sobel

As shown in the figures above, the Scharr operators only would produce edges in one direction at a time and the Sobel operator did not produce distinct edges. Thus we ruled out using the gradient edge detector algorithms. For these reasons, we felt DoG and Canny both had their strengths and were both preferred by an equal number of test subjects. Thus we designed our app toallow the user to choose which edge detector algorithm they use and switch between the two in real-time.

5.2 Region Smoothing and Quantization

The next step was to choose a region smoothing algorithm from bilateral filtering and Gaussian smoothing. The results are shown in **Figure 7** below.

Figure 7 a) Bilateral Filter b) Gaussian Smoothing

As we had theorized, the Gaussian smoother does not retain the edges as well as the bilateral filter, and for this reason we used bilateral filtering for our algorithm. We ended up not doing iterative bilateral filtering however because the number of iterations had to be significant to make a noticeable difference on the image and this became too computationally expensive.

Next we took the bilateral filtered image and quantized the colors. As shown in **Figure 8** below, the quantizing was essential to the "cartoonization" process of the image.

Figure 8 Bilateral Filter and Quantization

The quantization effects are most prominent on consistent color regions, such as the white wall or the black phone. Certain regions were quantized to one shade while other regions took on another shade.

5.3 Final Images

The last step to creating cel shaded images was overlay the edge maps over the quantized images. The results turned out well using both Canny and DoG edge detection along with bilateral filtering and quantizing.

Figure 9 Cel Shaded Image with Canny Edge Detector

Figure 10 Cel Shaded Image with DoG Edge Detector

5.4 Division of Labor

There were two major parts to this project: research and implementation. We were able to each focus on one of these in order to best utilize our skills. Sharud Agarwal focused on research of methods and related projects while Helen Kuo focused on implementation because of her prior experience with Android development. We both

however were able to contribute to all phases of the project and increase our understanding of the problem we were tackling.

6. CONCLUSION

The final results using the Canny or DoG edge detector algorithms turned out well in cel shading the images. The algorithms do tend to struggle with faces, however, we have had more consistent success taking pictures of sides of faces and we have even had a few portrait shots turn out well. We feel we were able to achieve our goal of giving consumers the ability to take cel shaded pictures from their mobile devices that do not take much processing time but also are appealing to look it and unique.

This project taught us a lot about the different filters we used and how also how heuristic filters can be used as add-ons to the main filters implemented. For example, with the edge detection algorithms, we spent a lot of time using open-close algorithms to link disconnected edges and also dilating to increase the widths of the edges. In addition, we found equalization (histogram stretching) to be very useful in edge detection as for some algorithms, the images would not be using the full pixel value range which comprised the algorithms edge detection capabilities. Overall, we are pleased with our outcome and the amount of knowledge we gained from this project.

7. REFERENCES

[1] Andrew Beaujon. "Pew: 82% use cell phones to take pictures." Internet: http://www.poynter.org/latest-news/mediawire/196207/pew-82-use-cell-phones-to-take-pictures/, Nov. 26, 2012 [Nov. 30, 2013].

[2] Henry Kang, Seungyong Lee, Chui C.K. "Flow-Based Image Abstraction". IEEE Transactions on Visualization and Computer Graphics, vol.15, pp. 62-76, May 16, 2008.

[3] Alan Bovik. EE 371R. Class Lecture, Topic: "Image Processing." ENS 115, Fall 2013.

[4] R. Fisher, S. Perkins, A. Walker, and E. Wolfart. "Gaussian Smoothing." Internet: http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm, 2003 [Dec. 2 2013].

[5] Kerry D. Wong. "Canny Edge Detection Auto Thresholding." Internet: http://www.kerrywong.com/2009/05/07/canny-edge-detection-auto-thresholding/, May 7, 2009 [Nov. 4, 2013].

[6] Andrey Pavlenko. "OpenCV for Android," ECCV, 2012.