Toonify: Cartoon Photo Effect Application

Kevin Dade
Department of Electrical Engineering
Stanford University
Stanford, CA
Email: kdade@stanford.edu

***Abstract*—Toonify seeks to emulate the types of cel-shading effects offered by graphics engines in a lighthearted and user-friendly way.**

I. Introduction

With the recent success of Instagram, the popularity of simple and fun photo effects apps has been on the rise. The mobile platform presents a unique arena for these applications by connecting users with both the means to capture images, and the computational power to perform sophisticated processing on these images. Toonify seeks to leverage the existing OpenCV libary for Android in order to emulate a particular effect known as cel-shading in the computer graphics world.

A. Cel-Shading

Cel-Shading is an effect typically associated with the rendering process of 3D graphics. The effect is achieved by first quantizing the color shades so that the shadows cast on an object falling under continuous lighting will form discrete areas of color. Secondly, contour lines may be added to accentuate the outlines of the model. This effect is relatively simple to achieve when the output is being rendered from models and light sources within the engine, but recreating the effect is not so easy with an arbitrary 2D input image. Due to the non-uniform lighting in a camera image, and the possibility for noise, it is difficult to acheive perfectly uniform color regions and outlines such as those afforded by the rendering process. Figure 1 shows an example of cel-shading rendered by a graphics engine.

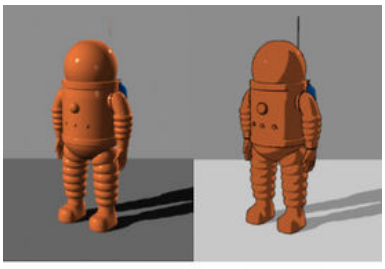

Fig.1

B. OpenCV4Android

The Open Computer Vision Library (OpenCV) provides a standard toolkit for performing basic and complex image processing algorithms. OpenCV4Android provides an Android-friendly interface to the OpenCV library, while still allowing the core classes of the library to take advantage of whatever hardware the phone has to offer. This allows mobile devices to complete computation-intensive tasks in relatively small amounts of time, which in turn allows for computationally expensive photo effects such as the cartoon effect described in this report. Because the cartoon effect relies heavily on a technique known as bilateral filtering, which requires a lot of computational resources, this user-friendly application is made possible by the OpenCV4Android package.

C. Project Goals

The goal of this project is to create an Android app with a simple user interface allowing users to apply the cartoon algorithm to images of their choice. The algorithm is designed to provide artistically and comically appealing results on as wide a range of pictures as possible, although it is conceded that not all inputs will yield equally satisfying results.

II. The Algorithm

The process to produce the cartoon effect is divided into two branches- one for detecting and boldening the edges, and one for smoothing and quantizing the colors in the image. At the end, the resulting images are combined to achieve the effect.

A. Edges

Finding smooth continuous contours is an important component to achieving the overall effect. The following steps are taken to provide contour detection that works in an artistically pleasing manner on a wide variety of images. All edge processing tasks are performed with a single-channel grayscale image derived from the luminance values of the input.

1) Median Filter: Before any further processing, a median filter is applied in order to reduce any salt and pepper noise that may be in the image. At this stage, the median filter kernel is a 7×7 matrix with the centroid as the center element in the matrix. The RGB pixel values at the centroid are set to be the median of the 49 RGB pixel values in the neighborhood. In this way, any extreme specks are smoothed over. The median filter is small enough to preserve edges in the image, which is important. Too large a kernel size would result in washed out edges.

2) Edge Detection: This algorithm uses the popular Canny edge detector. The benefits of using the Canny edge detector instead of a Laplacian kernel is that the edges are all single

pixel edges in the resulting image. This allows for morphological operators to be employed more predictably on the resulting edge image.

3) Morphological Operations: Currently, the only morphological operation employed by the algorithm at this stage is dilation with a small 2×2 structuring element. The purpose of this step is to both bolden and smooth the contours of the edges slightly. Different combinations of morphological operators are still being tested in this stage, and the aesthetic of the contours in the final image depend largely on this stage.

4) Edge Filter: Finally, the edge image is separated into it's constituent regions, and any region with an area below a certain threshold is removed. In this way, small contours picked up by the Canny edge detector are ignored in the final image, which helps reduce unwanted line clutter in the result. Empirical testing with the minimum area threshold showed that a minimum value of 10 seemed to produce the most consistently desirable results across the sample images.

B. Colors

The other important aspect of the cartoon effect is that of blockish color regions. In this branch of the algorithm, the colors are repeatedly smoothed to create homogenous color regions. The colors in these regions are then requantized at a lower quantization.

1) Bilateral Filter: This filter is the key element in the color image processing chain, as it homogenizes color regions while preserving edges, even over multiple iterations. Because it is a computationally expensive task, the image is downsampled by a factor of 4 in both the x and y directions before being filtered. The bilateral filter works similarly to a Gaussian filter in that it assigns to each pixel a weighted sum of the pixel values in the neighborhood. However, the difference is that the weights are further adjusted depending on how different the pixel values are. In this way, a pixel that is close in color to the centroid pixel will have a higher weight than a pixel at the same distance with a more distinct color. This extra step in the weight calculation is important because it means that sharp changes in color (edges) can be preserved, unlike with a simple Gaussian blur. However, the bilateral filter runtime is dependent on the kernel size, and testing showed that running more iterations with a smaller kernel yielded results that were more aesthetically pleasing, and faster than those yielded by applying a bilateral filter with a large kernel size with less iterations. The Toonify algorithm uses a 9×9 kernel and iterates the filter 14 times on the image. Once the filtering is complete, the image is restored to it's original size using linear interpolation to fill-in the missing pixels.

2) Median Filter: A median filter as described above is applied after restoring the image to its full size in order to smooth over any artifacts that occurred during the upsampling. The kernel size is 7×7, as before, and the effect is hardly noticeable, except in some regions where there are tiny islands of a certain color located in a larger homogenous region of a different color.

3) Quantize Colors: The final step in the color image processing chain is to requantize the color palette of the image. This is achieved with each color channel by applying the following formula, where p is the pixel value, and a is the factor by which the number of colors in each channel is to be reduced:

$$p_{new} = \lfloor \frac{p}{a} \rfloor \times a$$

Admittedly, this has the unwanted effect of truncating colors with pixel values close to the maximum, but in practice the absence of max-value colors is hardly noticeable. In this algorithm, a scale of $a = 24$ was chosen. This means that the standard RGB color palette of 256^3 colors is reduced to a palette of $[\frac{256}{24}]^3$ unique colors. Upon testing, this scale factor produced the most desirable results.

C. Recombine

Once both the color and edge image processing chains are complete, the only task left is to overlay the edges onto the color image. It is possible to draw the edges in different colors on the image, and during development an algorithm was tested wherein each edge is drawn with a color based on its immediate surroundings, but this yielded unpredictable and aesthetically unappealing results. The final algorithm simply draws on all the contours in black.

III. Results

As with any algorithm that seeks to meet an aesthetic goal, measuring the success of Toonify poses some difficult problems. Specific goals throughout the process were to achieve solid continuous contours while avoiding small line clutter in the image, and also to achieve large regions of homogenous color, with a reduced color palette. Output images are evaluated on how well they meet these criteria. The following figures contain examples of desirable outputs and a result that does not meet the criteria. Figure 2 shows

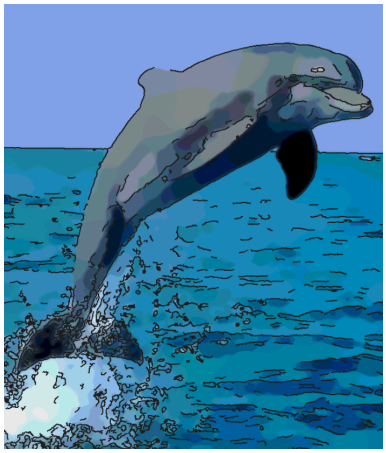
Fig.2

perhaps the best example of what the algorithm can produce. Notice that the sky has been turned into a single block of color, and that the dolphin and splash parts of the image have all the important contours drawn while avoiding any excess

line clutter. This is ideally the effect that Toonify attempts to deliver. In the example in figure 3, the colors are slightly

Fig.3 

more varied to begin with, and the lighting is even more non-uniform than in the previous example. However, the color blocks are mostly homogenous, and the contours mostly follow the contours that a viewer would associate naturally to the image. The contours in the sky along the cloud borders are not desirable. Finally, figure 4 shows an example of where the

Fig.4 

algorithm fails. The algorithm is particularly bad at handling portrait images in a natural way. This is because the face casts many small shadows that a human artist would not typically draw contours around. However, the Toonify algorithm picks them up anyway.

IV. Conclusion

All things considered, the Toonify algorithm is a success. It is capable of producing exactly the effect specified above, and a wide range of input images will yield satisfactory results. However, the algorithm is not perfect, and it does not respond predictably across all inputs. Given the variety of input images, it would be unrealistic to expect a one-size-fits-all approach to produce consistent results. In the future, an adaptive algorithm would probably be better suited to providing a consistent effect. Such an algorithm would recognize image types (portrait, indoor scene, outdoor scene for instance) and then change the edge detection parameters to better suit that particular image. Additionally, the algorithm may rely on an image derived from the hue, rather than the luminance, for edge detection. This might aid the algorithm in drawing contours around regions of different colors. Currently, the algorithm seems best suited for images that have a few large regions, each with relatively low color diversity. Typically the images that fail are those with a high amount of local color variation and detail.

Acknowledgment

The author would like to thank the instructors of EE368 for teaching a great class, and for providing useful resources throughout this project.

References

[1] Baggio, Daniel L, *Mastering OpenCV with Practical Computer Vision Projects* Packt Publishing Ltd, 2012