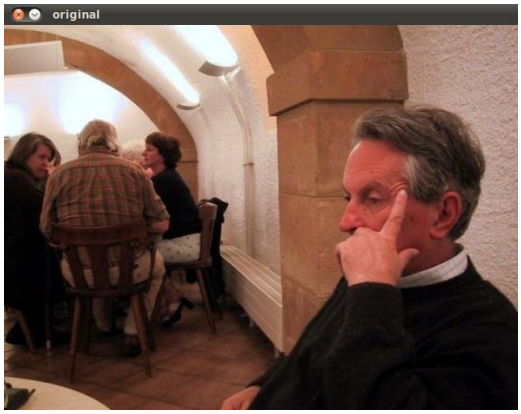
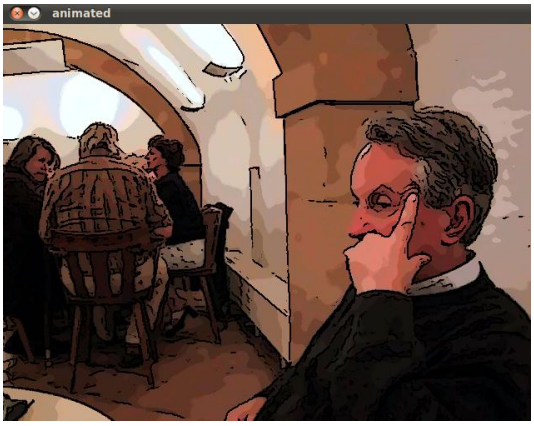
Artistic Filtering

By Zechariah Dzegede (zkd2) Peng Wei (pw272)





Introduction

Non photo-realistic rendering is a branch of computer vision that focuses on artistic expression rather than realism. NPR has been used in movies and games to produce a unique look and feel.

Image 1. A screenshot from the movie *A Scanner Darkly*

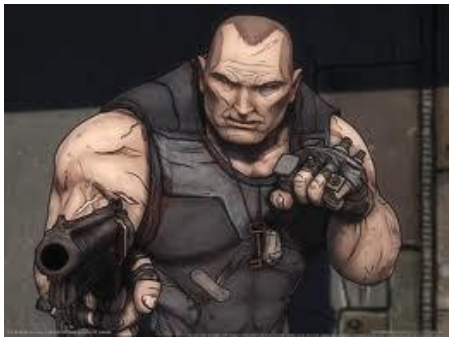


Image 2. A screen shot from the game *Borderlands*

Our goal is to create an image filter that takes a photo/video and converts it to an image/video that has the same feel as the above examples.

Related Work

We implemented the the algorithm described in the paper “Real-Time Video Abstraction” by Hoger Winnemoller et. al (http://www.cs.northwestern.edu/~holger/Research/papers/videoabstraction.pdf).

Algorithm

The algorithm we implemented consists of 3 steps:

1. Abstraction
2. Edge Detection
3. Quantization

Abstraction

Our approach for getting a cartoony feeling is to remove the high frequency details of a photo. We can do this by blurring the image. However, if we do a naïve blur like Gaussian, we will lose the edges of the photo. Hence, we need to use the bilateral filter. In the beginning we implemented our own

$$H(\hat{x},\sigma_d,\sigma_r)=\frac{\int e^{-\frac{1}{2}\left(\frac{\|\hat{x}-x\|}{\sigma_d}\right)^2} w(x,\hat{x}) f(x)\,dx}{\int e^{-\frac{1}{2}\left(\frac{\|\hat{x}-x\|}{\sigma_d}\right)^2} w(x,\hat{x})\,dx}$$

$$w(x,\hat{x},\sigma_r) = (1-m(\hat{x}))\cdot w'(x,\hat{x},\sigma_r)+m(\hat{x})\cdot u(\hat{x})$$

$$w'(x,\hat{x},\sigma_r) = e^{-\frac{1}{2}\left(\frac{\|f(\hat{x})-f(x)\|}{\sigma_r}\right)^2}$$

bilateral filter by using the following set of equations (we ignored the m terms).
But this turns out to be too slow so we used the opencv function cvSmooth(src,dst,CV_BILATERAL).

Instead of filtering the image in its original color space (RGB), we converted it to the LAB color space and performed the abstraction on the L (luminance) channel, so that all of the abstraction was performed only on luminance differences.

Edge Detection

We implemented edge detection as a modified version of DoG. After the image had been abstracted, we used DoG to extract the edges. The edges were needed because cartooned images have strong edges that help distinguish boundaries clearly, making objects seem more defined. The strongest parts of the edges took on the highest value, 255. Rather than making the image have 255 or 0, we followed the referenced paper's suggestion and tapered off the edges with a *tanh*, which causes them to lightly blend into the rest of the image.

Quantization

The next step is to quantize the abstracted image. The purpose of this is to simplify the representation of luminance so that it jumps in discreet steps. If you pay close attention to the luminance in cartoons you will notice that this same effect is present, simplifying the view of the object in terms of its luminance.

Each pixel originally has 256 luminance values (it was 100 values when in float, but became 256 values when we converted to uchar), and we reduce it to n bins so that the first 256/n values become one value, the second 256/n values become the second value, etc.

Essentially we used the following equation:

Quantized value = round(original value/(256/n))*(256/n), where n is the number of bins.

We used 8 bins in our code. In the sky in image 4 you will notice clearly how these bins interact.

Combining to get Results

The last step is to combine the abstracted L channel with quantization, the A&B channels, and the edges to produce a final image.

Image 3. Original image (top left), abstracted L channel (top right), edges from the abstracted L channel (bottom left), final combined image(bottom right).

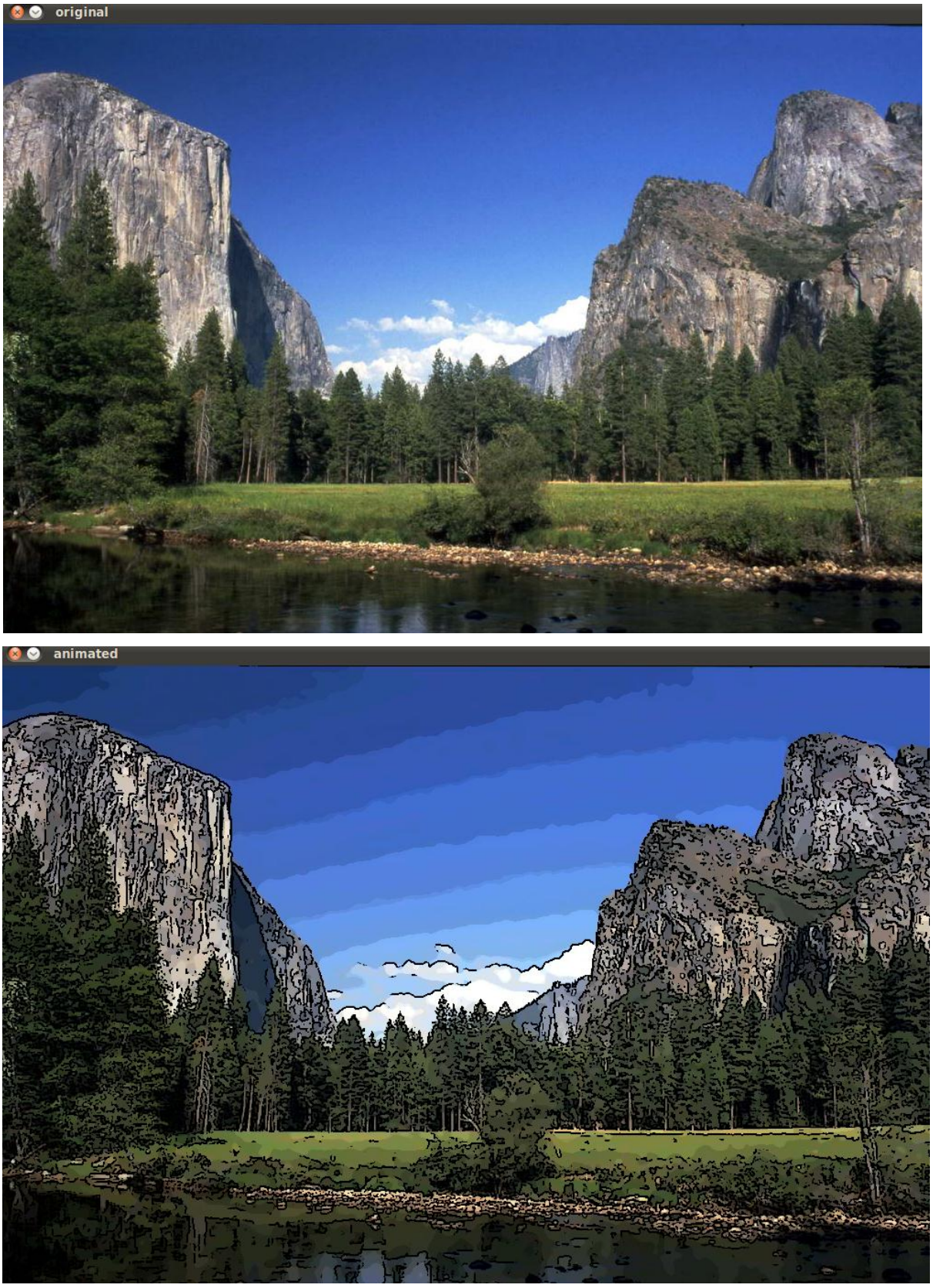


Image 4. Original (top) vs filtered (bottom) of a scenery.

Image 5. Original (left) vs filtered(right) of a group of people.

Image 6. Original (left) vs filtered(right) of a cartoon.

Discussion

Our filter worked well on the computer. It reliably abstracted the image and detected edges as shown in image 3,4,5. It did nothing to a picture that's already a cartoon, which is expected. The program ran in less than a second for an image with size (800*600). However, when ported to the phone, the speed started to drastically decrease; to process one frame took about 10 seconds. Hence, we needed to compromise quality of our filter to make things faster. Instead of using a bilateral filter with kernel size 7*7, we had to use a kernel size 3*3, which resulted in a loss of some uniform patches and edges as can be seen from the video taken on the phone.

When we tried to filter videos frame by frame, reducing the kernel size alone doesn't increase the speed enough. Hence we change our implementation of edge detection and quantization to make them work on 8-bit uchar images instead of 32-bit float images. Again, this did speed things up, but the quality and reliability of edge detection was lowered. The loss of edges is evident in our video. Even after all the optimization, we were only able to get a frame rate of about 1 fps or less, if we used an image that

was 320*240.

Our overall goal was to use video on the phone, so while setting up the code to work faster we neglected to keep the pictures we took in the intermediate processes (when we hadn't yet implemented video). I have also included in a folder called "videos" a video from the phone with a 640*480 resolution. To make things work faster we would have used 320*240 but we didn't because those videos could not be played back once we extracted it from the phone, for some reason). You will also notice that things looks a bit sped up from reality. This is because it took so long to process each image, which the cartoon code would only act on every couple frames. By making the video run at about 10fps we end up getting a video that took a couple minutes to make, but lasts on the order of 10 seconds on playback. In the included video I have slowed down the phone videos by 0.75. Also there is a video of breakdancing which we performed abstraction on using the computer. Once again this was after we converted our float images to uchar, so that edges are not as prominent.

We would also like to note that in trying to make the video quality look better we messed with parameters, our results seen in this report show the best results for pictures using certain parameters. Right now the code does not have as good parameters (since changing the kernel size affected the parameters), mixed with the fact that the 3*3 kernel works worse anyway, and the fact that we are now using uchar instead of floats, the quality is much worse. We didn't feel the need to waste time trying to find the perfect parameters for the sake of turning in our code

Future work

The speed of our program can be improved by integrating OpenGL. If speed is faster, then we don't have to compromise quality as much, which may lead to stronger edges and more uniform patches, hence an overall better filtered video.

Who did what

Zack wrote the first run of the code, implementing all of the 3 steps. He also contributed to figuring out how to compress the code, as well as implementing the compression. He handled making the videos and taking some of the pictures. Zack revised this report and added new information after Peng did the initial write up.

Peng figured out how to do abstraction using opencv. He also contributed to figuring out how to compress the code, as well as implementing the compression. Peng wrote the first run of this report, and worked on getting the good PC based results you see in this report.