Matthew Pana

Professor Ming-Ting Sun

EE 440, Fall 2023

EE 440: Final Project

Introduction

This document is an accompanying report for the Final Project in EE 440 Introduction to Digital Imaging Systems. The purpose of this project is to show our proficiency in implementing some of the techniques learned in class. A personal purpose is also to implement some post-processing techniques I have seen outside of class that I was personally interested in.

This project has two parts. The first is creating a GUI that is able to display two images side by side, one with a filter applied to it. The GUI should also allow the user to change which filter is being applied. The second part is choosing which images to use and creating any number of filters to be applied to them.

Before starting, I knew I wanted my GUI to use HTML so that I could potentially more easily use the project in some sort of portfolio. Additionally, I knew before I had implemented any filters that I wanted to try my hand at making a Kuwahara filter, which I was introduced to in a YouTube video.

Instruction on How to Run

The project is accessible at the following URL: https://barbarianmatt.github.io/filterFinalProject/
The project was made with Firefox in mind, though I do believe that it should work on any other browser on any device. I backup zip file is also provided in case you have a lack of internet connection or some problem with the webpage was on my end. After unzipping the folder, you will see 3 folders and 3 files. Open the index.html file and it should open a local page on your browser. Everything else will be the same as if you opened it from the URL.

GUI Description

Immediately upon opening the URL, you will be greeted by two Lena images, a -filter select- button, a -change image- button, and a -toggle looking glass- button. The image on the left is the original and the image on the right is the original after the currently selected filter is applied. If you hover over an image, a zoomed-in looking glass of the pixels near the tip of the mouse will be displayed. You can tell an image is still loading if the looking glass does not appear. The looking glass can be toggled off by pressing the -toggle looking glass- button. Clicking the select filter button brings up a menu that allows you to select other filters. Clicking the change image icon cycles the original image between Lena, Rose, and Bladerunner. Underneath the select filter button are individual filter parameter sliders, which change based on the filter currently selected. They allow you to change the parameters of the currently selected filter in real time.

GUI Implementation

The homepage of the GUI is a simple html file with basic container and button elements. The CSS was based on a previous project I did but with a different color scheme. The looking glass is a canvas of 30x30 pixels whose values are adjusted to the same as where the mouse is currently located, by indexing the image file. The -change image- button changes the source of the image HTML object between the 3 I already picked out. I picked Lena because it is the most popular image-processing example. The particular image of a rose was one I found that I liked because the somewhat random background is a good test for whether filters create color banding or have any other weird effects with this out of the ordinary background. The third image I chose was a capture from the movie Blade Runner 2049. I felt it was a good test on how filters reacted to dark but not homogenous backgrounds and it has a nice constant of color with next to a distinct silhouette. The rest of the GUI has mundane implementation details so I will choose to skip it.

Filter Descriptions

I implemented 9 filters, Grayscale, Invert, Gaussian Blur, Median, K-Means Segmentation, Modified Generalized Kuwahara, Quantization with Dithering, Quantization with Random OkLab color palette, and the Actual Generalized Kuwahara Filter. The first four I would classify as standard filters, and the last 5 I would classify as artistic. Kuwahara, dithering, and OkLab palette creating were concepts taken outside of the class, while the rest of the filters/concepts were talked about in class.

Filter Implementation

Grayscale

Grayscale has only 1 parameter, richness. This is a binary indicator tells the filter whether to use rich grayscale or standard. The idea for rich vs standard was that through my research, I found two alternative ways of getting the grayscale. One was the average of the R,G,B and the other was dotting the RGB with the vector (0.21,0.72,0.07). I decided to implement both in my implementation.

Invert

Invert has no parameters. It goes through every pixel and calculates the inverse of the RGB value using the formula $255 - RGB$.

Gaussian Blur

Gaussian Blur has 1 parameter, radius. This is an integer that ranges from 0 to 10. Gaussian blur is calculated by first creating a kernel with weight based on the Gaussian function with side length 2 times radius. Then it iterates over all pixels of the image and gets a new pixel by adding up the neighboring pixels using the Gaussian kernel. Pixels outside of the image are ignored instead of replaced with neighboring pixels which is the standard. This choice was made because JavaScript is not a very good language for image processing so I wanted to save computation time, an issue that will be more prevalent as we go on.

Median

Median has 1 parameter, radius. This is an integer that ranges from 0 to 10. The median is calculated by iterating over each pixel of the image and putting all neighboring pixels in a square of side length 2 times radius into an array. It then sorts that array to find the median, which it uses as the new pixel.

K-Means Segmentations

K-Means Segmentations has 1 parameter, regions. This is an integer that ranges from 2 to 10 and represents how many regions the K-Means algorithm should use. First, the image is converted to rich grayscale. Then, the standard K-means algorithm is applied, first, some number of region center points are created. Then each pixel is assigned to its closest region, where distance is the difference between its grayscale value and the grayscale value of the center of the region. Then, after all assignments, the center of each region is adjusted based on the average of the pixels in that region. This is repeated 100 times. Then each region is assigned a color. Originally, this was random but I decided later on to borrow the random color palette code from the Quantization with Random OkLab color palette filter to make this one look better. The K-Means algorithm could have instead used RGB instead of grayscale, however after trying this the results were not that different and the performance was obviously 3 times worse so I reverted back to grayscale.

My Modified Kuwahara Filter

This filter has 3 parameters, Radius, Granule size, and hardness. Radius is an integer that ranges from 0 to 10 and controls how big the Kuwahara kernel is. Hardness is a float ranging from 0 to 20 that attempts to control how large continuous areas blend with each other. Granule Size is a float that ranges from 0 to 1 and attempts to control how much color variation there is inside continuous areas of similar color. My implementation was based on (Papari G., 2007). To be specific, it is the generalized Kuwahara filter, not the Anisotropic Kuwahara filter. Neither of these terms are on the paper as they were coined later. I also looked at (Kyprianidis J., 2010) for help in implementation. After implementation, the algorithm was very slow, mostly due to the limitations of JavaScript. So instead, I modified the algorithm and got rid of any Gaussian weighting. The following is my modified implementation. First, I create a circular kernel and a sector kernel. The circular kernel is square such that any points with a distance less than or equal to the radius parameter are 1, and everything else is 0. The sector kernel creates a square and assigns each point a number 1-8 depending on its angle i.e. splitting the kernel into 8 radially symmetric sectors. Then we iterate through each pixel of the image. For each pixel, ignore the neighbors outside the circular kernel. Each neighbor will be in one and only one sector, so for each sector, find the average and standard deviation of the RGB of the pixels in that sector. Each sector also includes the original i.e. center pixel. With these means and standard deviations, create a weight $w_i = \frac{1}{1+(q\cdot\sigma_i)^{\frac{h}{2}}}$ where h is the hardness parameter and q is the granule size parameter. Using these weights, which are $w_i \in \mathbb{R}^3$, we get the new color value $c = \frac{\sum w_i m_i}{\sum w_i}$ where m_i are the means we calculated earlier.

Quantization with Dithering

This filter has 3 parameters, Levels, Dither Size, and Spread. Levels is an integer ranging from 2 to 16 that determine how many different values a single element of a color can be. i.e. if Levels is 2, the maximum amount of different colors is $2^3 = 8$. Dither size is an integer ranging from 1 to 3 and controls how large the dither texture is. Spread is a float ranging from 0 to 0.5 and controls how much of an effect dither has. First, a dither texture is created with size depending on Dither Size. The dither textures are premade and have values pseudo-randomly placed ranging from 0 to $n^2 - 1$ where n is the side length of the texture. The dither value is calculated from the equation $s\,\frac{ditherTexture[x\%n][y\%n]-0.5}{n^2} \cdot 256$ where s is the Spread. Dither acts as a kind of controlled noise to make the boundary between colors better on the eyes by eliminating color banding. This dither value is added to the original RGB. Then the R value is quantized by the equation $\frac{\left\lfloor \frac{R}{256}(l-1)+0.5 \right\rfloor}{(l-1)} \cdot 256$ where l is the Levels parameter. This value is then clapped using floor, min, and max to ensure it is a valid pixel value. This is repeated from G and B and the new pixel value is just the combination of all three.

Quantizing with Random OKLab Palette

This filter has 1 parameter, Levels. Levels is an integer ranging from 2 to 16 that determines how many different values a single element of a color can be. This filter is the same as the previous filter except I have the dithering size and spread pre-chosen and colors are mapped to a custom color palette at the end. This is done by converting the image to rich grayscale before quantizing and then mapping the quantized grayscale values to a generated OKLab palette. Using randomly generated seed values as settings, I use the OKLab color space to generate equally spaced points whose hues and luminances increase, while saturation stays the same. This is then converted back to RGB for actual use. I originally used randomly generated colors, but the result did not look great. So after some research on how to make better palettes, I came across this approach.

Actual Generalized Kuwahara

This filter has 3 parameters, Kernel Size, Granule Size, and Hardness. The three parameters have the same effect as they do my modified filter. This filter is a JavaScript implementation of a Kuwahara shader I found on GitHub (Gunnell, n.d.). The purpose of this implementation is to show the difference between my Modified Filter and the actual Generalized Kuwahara Filter.

Results Discussion

I will not talk about the standard filters as their outputs and effects of parameters were explained in class and are obvious.

K-Means Segmentations

The load time is ~2 seconds. Having fewer regions, I believe, makes the result look better. This is because, with higher region counts, the regions intersect frequently leading to messy parts of the images. This can be seen in Figure 1. K-Means looks worse compared to Quantization with a color palette, even though they use the same color palette, because the regions in K-means have no reference when assigning colors. The colors are assigned arbitrarily. This is different compared to quantization, as the darker grayscale pixels get assigned to darker colors on the palette and vice versa for lighter pixels.

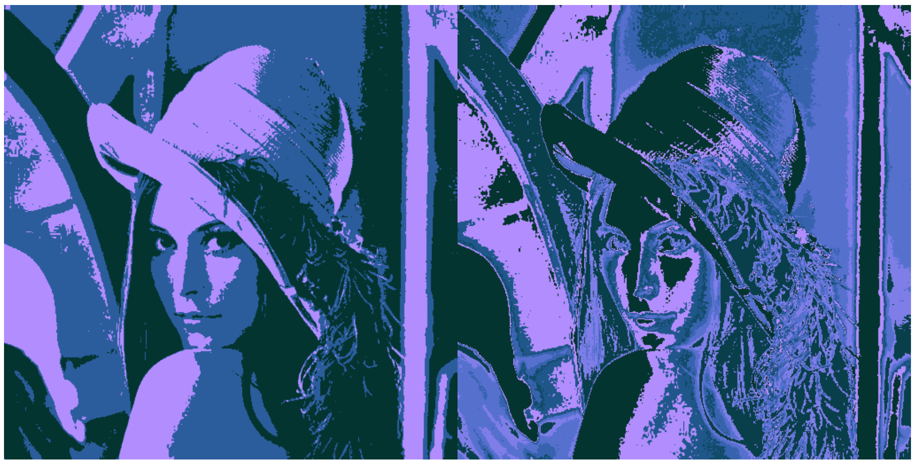

Figure 1: K-Means with 3 Regions vs 8 Regions

My Modified Kuwahara Filter

The load time is ~8 seconds. At higher kernel sizes, the painterly aesthetic of the filter becomes more apparent. However, areas of high detail become muddied. This is apparent in Figure 2 when you look at the eyes. At lower hardness, the blobs of color blur together at the edges, while at higher hardness, the blobs of color have very distinct edges as seen in Figure 3. Granule size is harder to see than the other parameters, but with lower granule size colors are more likely to mix at the edges. This can be seen on her forehead in Figure 4.

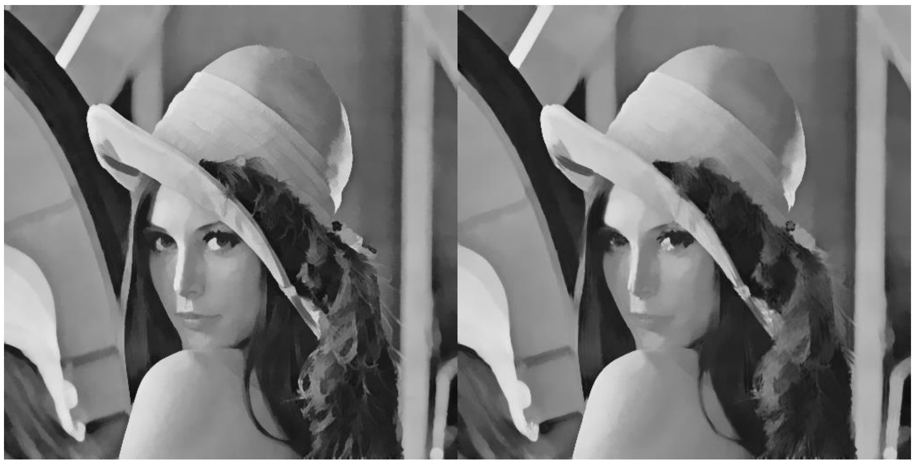

Figure 2: Modified Kuwahara Kernel 6 vs Kernel 9

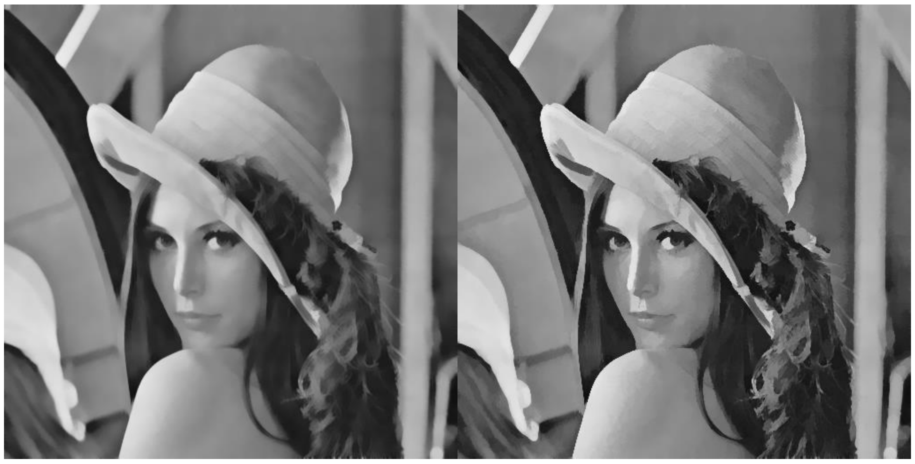

Figure 3: Modified Kuwahara Hardness 4 vs Hardness 16

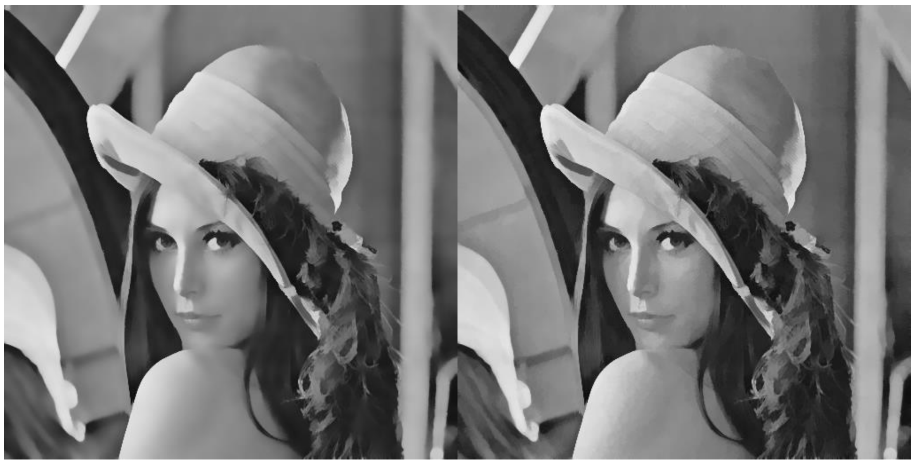

Figure 4: Modified Kuwahara Granule 0.1 vs Granule 0.8

Quantization with Dithering

The load time is <0.1 seconds. At a lower number of levels, the image quality is significantly reduced due to the lack of variety in colors. At a higher number of levels, the image looks very similar to the original, but with a retro aesthetic due to the still limited color palette, as shown in Figure 5. The dither size parameter, which controls whether the dither texture is a 2x2, 4x4, or 8x8 does not have that much of an effect on the image and is mostly preference, as shown in Figure 6. Low dither spread reintroduces color banding and makes the constrained color palette of the image more apparent. High dither spread makes the dither texture very obvious all over the image. Thus, it is best to choose a value in between. The difference in dither spread is shown in Figure 7.

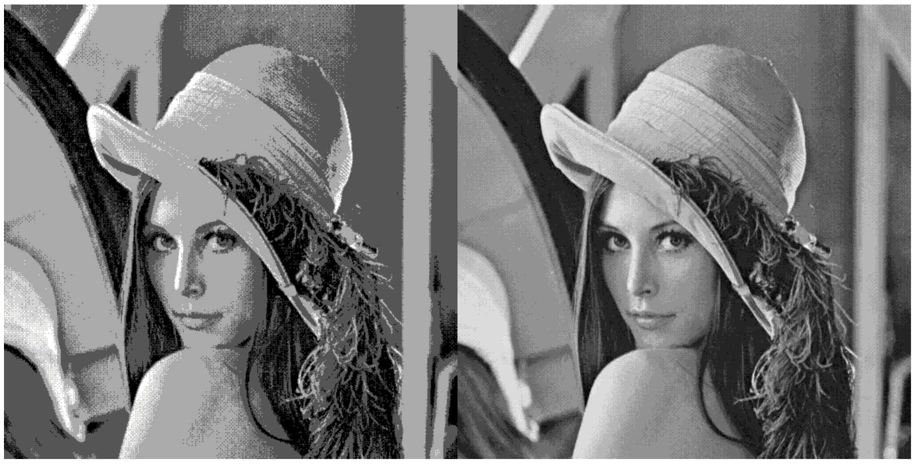

Figure 5: Quantization with Dithering Levels 4 vs Levels 8

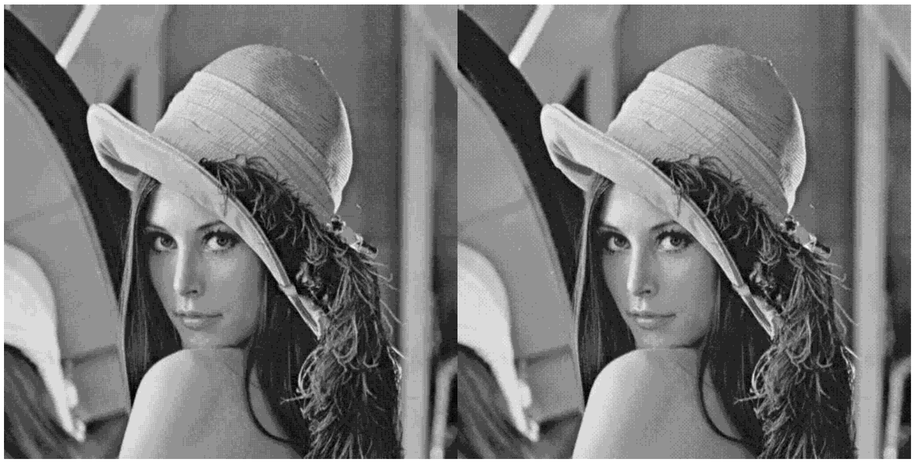

Figure 6: Quantization with Dithering Dither Size 1 vs Dither Size 3

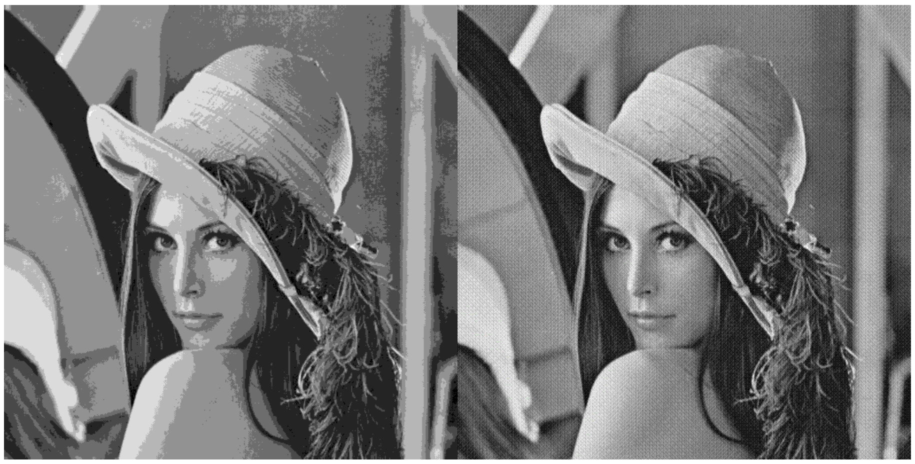

Figure 7: Quantization with Dithering Dither Spread 0.02 vs Dither Spread 0.45

Quantization with Random OKLab Palette

The load time is <0.1 seconds. Levels again controls the quantization of the image, but now also determines the exact number of colors used i.e. if Levels is 8 then the image is quantized to 8 colors which are then mapped to an 8-color palette. Now, lower number of levels doesn’t look as bad as before as the color palette makes the lack of colors look purposeful. However, this is not true for the images that were originally colored. The earlier filter condensed the colored images into n^3 colors. Now, the images are being compressed into just n colors. Unlike Lena, their grayscale counterparts don't stand out as much. Thus, when we apply the mapping to n colors, a significant amount of detail is sacrificed in comparison to the previous filter, which had a broader range of colors to preserve more details. Figures 8 and 9 show an example of what I am talking about.

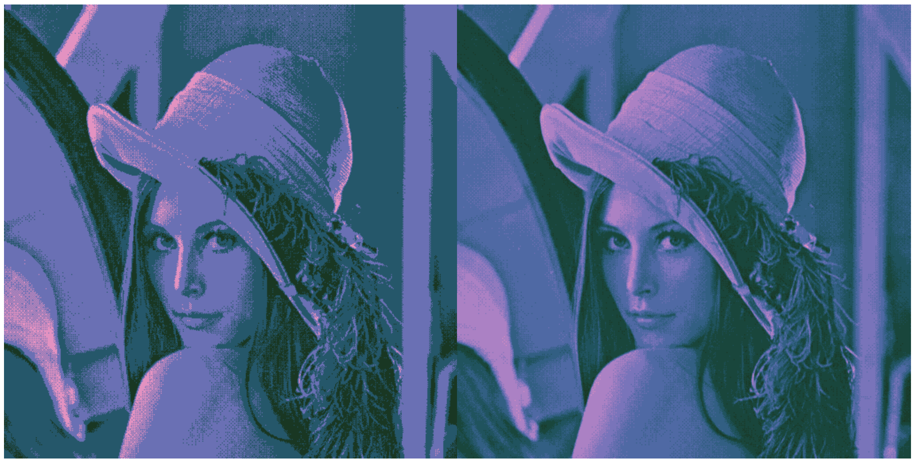
Figure 8: Quantization with Random OKLab Palette Lena Levels 4 vs Levels 8

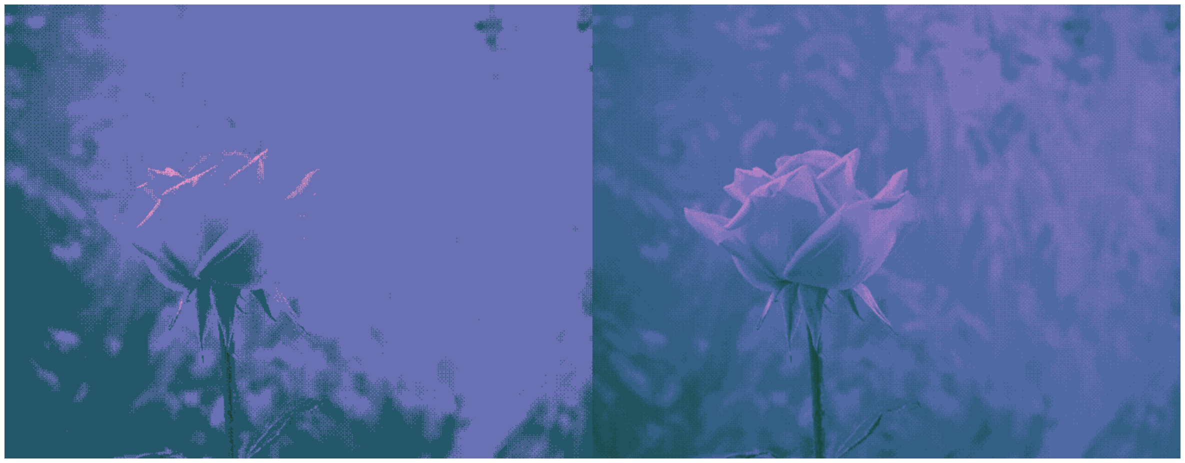
Figure 9: Quantization with Random OKLab Palette Rose Levels 4 vs Levels 8

Actual Generalized Kuwahara

The load time is 20-40 seconds. Again, this filter was implemented just so that I have something to compare my modified filter to. The load-time makes the filter impractical for trying to modify parameters in real time. Compared to the modified filter, the kernel size does not need to be as large to get similar effects e.g. Kernel Size 4 is similar to size 6 for the modified filter. Additionally, it seems to better preserve details at higher kernel sizes, as seen in Figure 10. Hardness again makes the edges between blobs of color more distinct, as seen in Figure 11. A major difference is that Granule size has little to no observable effect on the output image, as seen in Figure 12. Overall, the slight loss in quality from my Modified filter more than makes up for the faster loading time in my opinion.

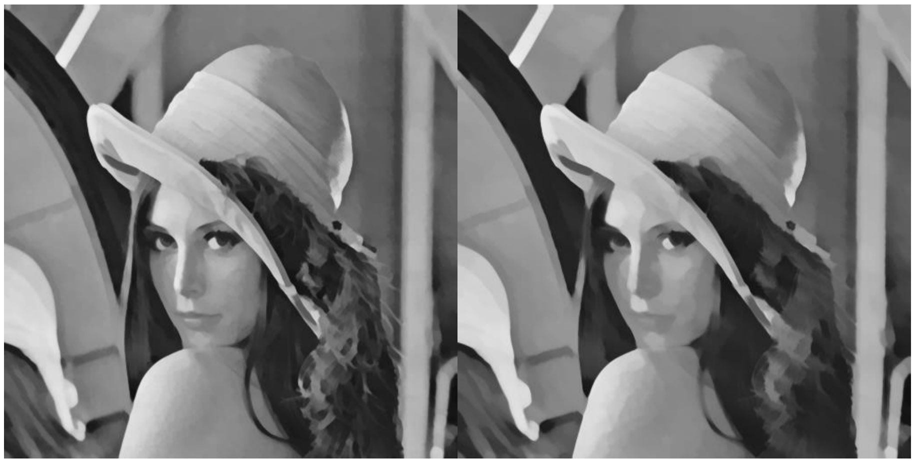

Figure 10: Generalized Kuwahara Kernel Size 4 vs Kernel Size 7

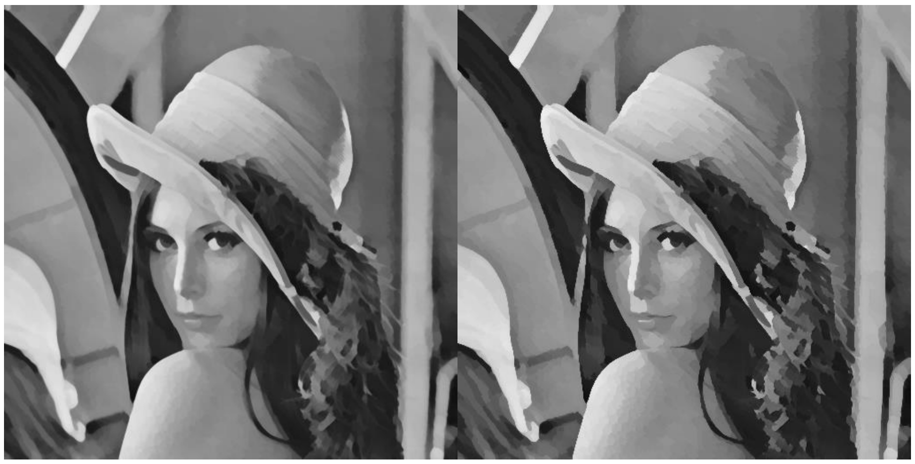

Figure 11: Generalized Kuwahara Hardness 4 vs Hardness 16

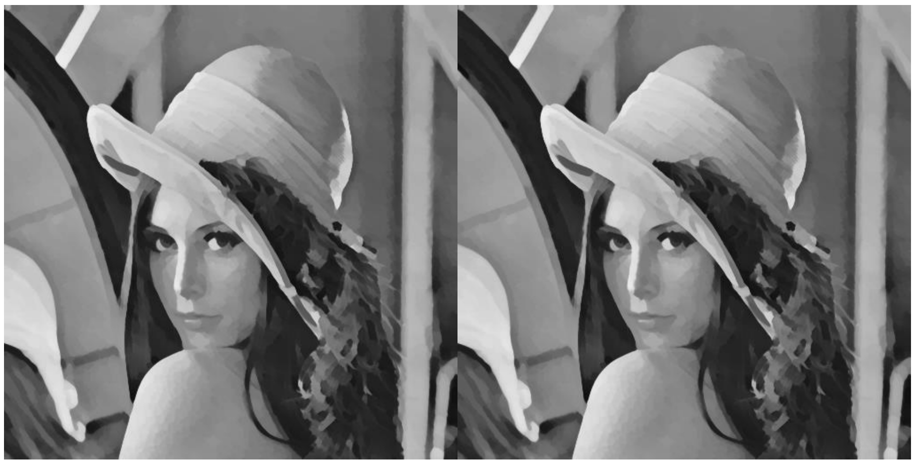

Figure 12: Generalized Kuwahara Granule Size 0.1 vs Granule Size 0.8

Conclusions

This project has highlighted some interesting concepts for me. The first is how difficult graphics programming is. When reading (Papari G., 2007), there were so many new concepts that I had not seen before, and trying to implement abstract mathematics with code was challenging. The second was that this was my first experience observing the distinct performance disparities among coding languages. Upon translating the Kuwahara filter code from C to JavaScript, its execution proved way slower than my initial expectations. This discrepancy was particularly obvious as the C code had to run once per frame in a game. Thus, my JavaScript code was ~200 times slower in comparison. If I did this project again, I would have coded up some backend such that a faster coding language would be calculating the filter instead of JavaScript.

References

Gunnell, G. (n.d.). *Post-Processing Repository*. Retrieved from Github: https://github.com/GarrettGunnell/Post-Processing/blob/main/Assets/Kuwahara%20Filter/GeneralizedKuwahara.shader

Kyprianidis J., S. A. (2010). Anisotropic Kuwahara Filtering with Polynomial Weighting Functions. *The Eurographics Association*.

Papari G., P. N. (2007). Artistic edge and corner enhancing smoothing. *IEEE Transactions on Image Processing*, 2449-2462.