FINAL PROJECT

1 Introduction

The past assignments have focused on aspects of image processing theory without delving into specific applications. This final course project will be exact opposite. We'll start with a problem statement and go from there. This reflects how image processing pipelines are developed in practice. Finally, it underscores how there often isn't a "correct" solution so much as a solution that best meets a set of functional requirements.

1.1 The Setup

Let's consider the following scenario. You're working for a small company that makes a suite of photo editing apps. Your team is responsible for the software framework that applies the various filters and effects within the app. This means that your team works with images directly; the framework's API consumes an image and produces the processed output.

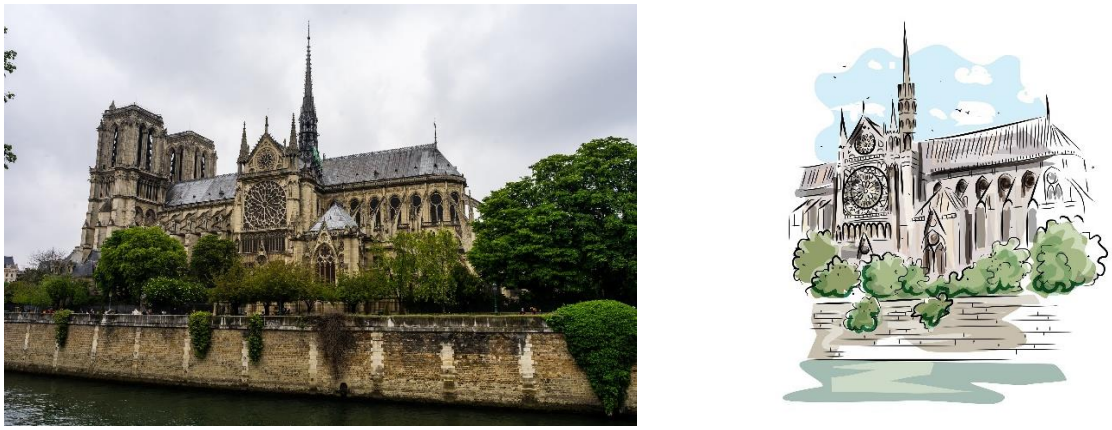

Figure 1 - A photo [1] and an illustration [2] of Notre Dame Cathedral in Paris, France.

During the Monday morning team meeting, your team lead announces that product management is looking to add a filter that can turn images into paintings. They're hoping to have this functionality in by the next release, which is only a couple of months away. You're all shown the "before/after" example (Figure 1) from an email sent by the project manager to the team lead late Sunday night.

This seems like a nearly impossible task. The team lead brings the team's concerns back to the product manager, who confesses to having watched A Scanner Darkly [3] over the weekend. It combined live action with animation to produce a surreal cartoon effect (Figure 2). That's what the style the new filter should produce. It's a much more reasonable goal than the original request. Armed with this knowledge, the team leads gathers the team to begin brainstorming and to start planning the work.

The point of all this framing is to help convey how the general idea of, "let's turn an image into a painting", becomes a piece of functional software. We'll use this scenario to guide how we build up the final algorithm and image processing software.

Figure 2 - Still frame from A Scanner Darkly. (Source IMDB)

1.2 Defining Requirements

Now, since this is a course project, we'll skip the whole process of negotiating what the effect should look like[1] and all the project management. We'll use Figure 2 as a rough starting point, keeping in mind that replicating this effect exactly using an automated process is *very difficult*.

We'll break this into two parts. First, we should note the following things about the overall image effect itself:

- Image edges are represented by black lines of varying thickness.
- There aren't many smooth gradients in the image.
- The overall appearance is "simple", meaning there isn't a lot of high-frequency texture.

Second, we should also note that this will be going into a larger software system. That means that it will have to meet a certain set of functional requirements for it to even be *used* by anyone else. Again, given this is a course project, and we'll keep things simple, meaning that the template project will contain a basic framework for you to work with.

Given all this, we can create a formal set of requirements[2]. They are as follows:

1. Stylization API
    1.1. Accept a single colour or monochrome image.
    1.2. A set of parameters to control the effect.
    1.3. Output the processed image.

[1] Depending on the size of the organization, and your seniority, this may involve negotiations with *many* different people. Collectively they're referred to as "stakeholders" because they have a stake in the outcome.
[2] The term "requirements" is being used to refer to a few different things. First, it defines what the stylization effect *must* do. Second, it describes *how* the effect interacts with other systems. In larger projects these will be often defined separately. Technically, this list is a combination of feature requirements and a functional specification.

2. Cartoon/Rotoscope Effect
    2.1. Image edges must be represented by black lines.
    2.2. The width of a line depends on its contrast or edge strength.
    2.3. The image should have less texture (high frequency regions) than the source image.
    2.4. Changes in luminance must come in discrete steps.
    2.5. The colour must be the same for a region with the same local luminance.
3. User Interaction
    3.1. The user runs the effect through a command line interface (CLI).
    3.2. Defaults are chosen to produce "good enough" results on a wide variety of images.
    3.3. The user *may* change the default value of one or more parameters.

The purpose of these requirements is twofold. One, it clearly states what is being delivered and two, it helps everyone know when the work is complete. Again, because this is a course assignment, some of these requirements will be fulfilled for you. Furthermore, we're going to invert the process a bit where rather than trying to develop an algorithm from scratch, you will need to show that the presented algorithm does in fact meet requirements 2.1 through 2.5.

2 The Algorithm

The formal algorithm is given in the supplemental material (Section 5.1). The high-level description is

1. Given an image $I(x, y)$:
    a. If the image is RGB, convert it into a greyscale image $I_g(x, y)$.
    b. Otherwise $I_g(x, y) = I(x, y)$
2. Compute the XDoG filtered image $E(x, y) = \mathrm{XDoG}\{I_g(x, y)|\sigma_x, p, T(I)\}$.
3. Compute the N-level level-set image $L(x, y)$ from $I_G(x, y)$.
4. Blur $I(x, y)$ with a Gaussian filter of kernel size σ_b.
5. If $I(x, y)$ is an RGB image, then:
    a. Find the connected components of $L(x, y)$.
    b. Set the colour of each component to be average colour when sampling from $I(x, y)$ to produce the shading image $C(x, y)$.
6. Otherwise, set $C(x, y) = L(x, y)/N$.
7. The final stylized output is $I_{stylized}(x, y) = C(x, y)E(x, y)$.

With this algorithm we can take a source image, like the one in Figure 3 and produce a stylized image like the one in Figure 4.

Figure 3 - Original image.

Figure 4 - Final, stylized image.

2.1 eXtended Difference of Gaussians

The algorithm is based around the eXtended Difference of Gaussians filter [4]. The XDoG filter is built on a relatively simple premise: use unsharp masking to boost "interesting" features in the image and apply a thresholding operator to produce a new, non-photorealistic image. The choice of parameters and thresholding can greatly affect the output.

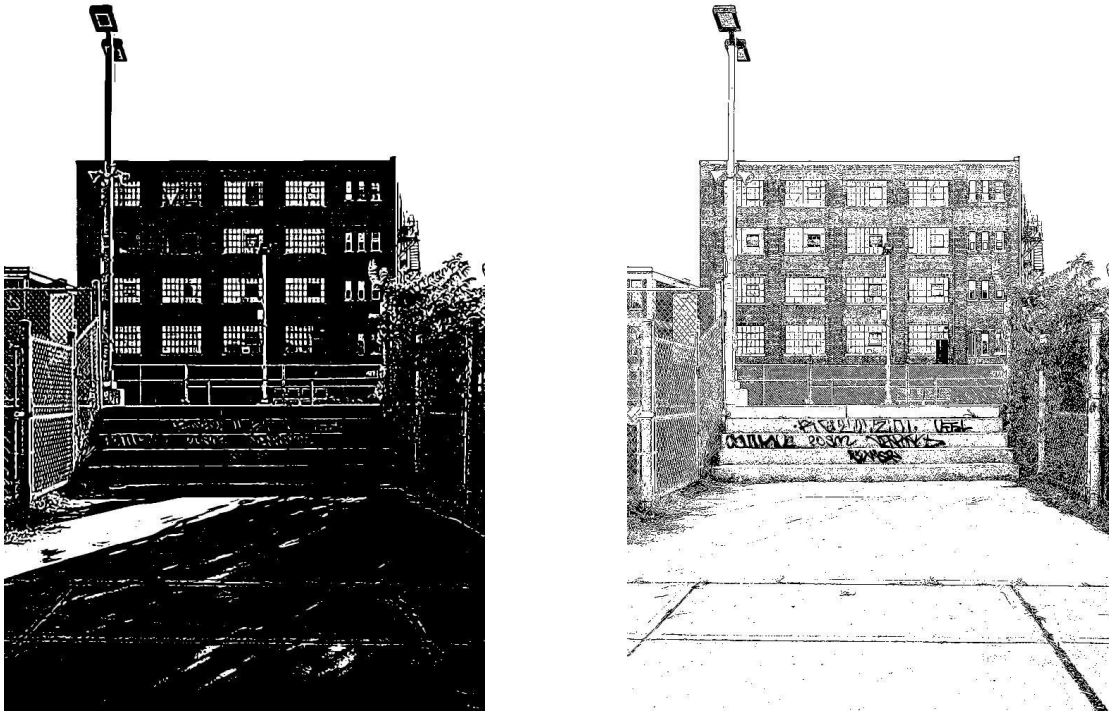

Figure 5 - Example of XDoG, as applied to Figure 3.

Take for example Figure 5. The *only* difference between them were the parameters for the XDoG filter when it was being applied to Figure 3. Even the same thresholding operator[3] was used! And yet, this is enough to produce two very different images.

One thing to note is that XDoG, as described in [4], contains an advanced filtering stage based on edge gradients. This can create a smoother edge appearance, though it's debateable on whether it creates a "better" effect. It's also somewhat tricky to implement, leading to a cost/benefit trade-off.

A simple way to improve the appearance of edges, especially when using a "hard" threshold is to find all corners and take the average value at that location. This has the effect of acting much like an anti-aliasing filter[4].

[3] Specifically, equation (3). See [4] for details.

[4] The term "anti-aliasing filter" isn't completely correct. This is really a way to simulate the effect of an anti-aliasing filter without having to subsample the image. Or, as used in computer graphics, to avoid having to render a very large image. See 22.6 (direct link) in [5] for an example of this.

2.2 Level Sets and Connected Components

A level set, as used for this algorithm, simply refers to the index of a N-bin histogram. In early assignments, the histograms that we generated matched the number of intensity values. However, there's nothing stopping us from using *fewer* bins. If we keep track of the bins, setting each pixel's intensity to the bin value, we end up with something like a topographic map.

Connected components is a type of graph-based algorithm that is used to quickly assign unique IDs to "blobs" within an image. For this assignment isn't not important to understand *how* it works so much as *what* it does. Specifically, imagine a binary image with a bunch of disconnected blobs. A connected components algorithm will label each one of these blobs with a unique IDs. The **`skimage.morphology.label`**[5] function in scikit-image is one such implementation.

3 Assignment

The assignment will be split into two parts (i.e. deliverables). The first part will focus just on the algorithm while the second will focus on creating the final, finished piece of software. The assignment breakdown is below. You are expected to consult with and demonstrate your in-progress assignment to your TA before the deadline (see course syllabus).

1. **[10 marks]** Present your in-progress work to your TA. This must be scheduled sometime before you submit the code and final report. If you are having difficulty implementing the algorithm, then use this time to consult with them to get feedback *before* the deadline.
2. **[10 marks]** Implement the assignment according to the specification in the template's README file. The included test runner will verify that it meets this specification.
3. **[35 marks (Total)]** Write a report describing what the algorithm is doing and how the various parts work together to provide the final, stylized output. You must consider the following:
    - **[10 marks]** What exactly is the XDoG filtering doing? How can it produce such different styling just by changing the filtering parameters? Provide examples.
    - **[5 marks]** The XDoG paper [4] describes a filtering step using an "edge tangent flow". How can this improve the appearance of the XDoG output? If you were to implement it, how would it be done?
    - **[10 marks]** What is the purpose of computing the level set? What happens if we skip that step?
    - **[5 marks]** What effect do you create when you set a pixel's intensity value to its histogram bin index? Consider a variety of values for N, e.g. 2, 5, 10, 50, etc.
    - **[5 marks]** What do we gain by applying the extra step of using connected components after generating the level sets?

[5] See: https://scikit-image.org/docs/stable/api/skimage.morphology.html#skimage.morphology.label

4 REFERENCES

[1] S. Wilson, "Notre Dame Paris France," 2017. [Online]. Available: https://pixabay.com/photos/notre-dame-paris-france-cathedral-4780725/.

[2] N. Lavrinenko, "Notredame de Paris Cathedral," 2019. [Online]. Available: https://pixabay.com/illustrations/notredame-de-paris-cathedral-4515298/.

[3] R. Linklater, Director, *A Scanner Darkly.* [Film]. United States of America: Warner Independent Pictures, 2006.

[4] H. Winnemöller, S. C. Olsen and J. E. Kyprianidis, "XDoG: An eXtended Difference-of-Gaussians Compendium including Advanced Image Stylization," *Computers & Graphics,* vol. 36, no. 6, pp. 740-753, 2012.

[5] E. Chan and F. Durand, "Fast Prefiltered Lines," in *GPU Gems 2*, Addison-Wesley, 2005, pp. 345-359.

5 Supplemental Material

5.1 Stylization Algorithm

Inputs:

- $I(x,y)$ – Input image (RGB or monochrome; floating-point).
- $\sigma_x, p, T(I)$ – XDoG parameters.
- σ_b – Gaussian blurring size.
- N – Number of levels to used when generating the level set.

Outputs:

- $I_{stylized}(x,y)$ – Output image (RGB or monochrome; floating-point)

Algorithm:

1. If $I(x,y)$ is RGB:
    a. $I_g(x,y) = Gaussian\{Rgb2Grey\{I(x,y)\}|\sigma_b\}$
2. Else:
    a. $I_g(x,y) = Gaussian\{I(x,y)|\sigma_b\}$
3. $I_{min} = \min\{I_g(x,y)\}$
4. $I_{max} = \max\{I_g(x,y)\}$
5. Allocate an array $bins$ of length $N+1$.
6. For $i \coloneqq 0$ to N do:
    a. $bins[i] = \frac{I_{max}-I_{min}}{N} i + I_{min}$
7. Allocate an image $L(x,y)$ with the same size as $I_g(x,y)$.
8. For $i \coloneqq 0$ to $N-1$ do:
    a. If this is the last iteration:
        i. $mask = I_g(x,y) \geq bins[i]$
    b. else:
        i. $mask = I_g(x,y) \geq bins[i] \ \wedge I_g(x,y) < bins[i+1]$
    c. $L(mask) = i$
9. Allocate an image $S(x,y)$ with the same size as $I(x,y)$.
10. If $I(x,y)$ is RGB:
    a. $\tilde{L}(x,y) = ConnectedComponents\{L(x,y)\}$
    b. Foreach label $L_n(x,y)$ in $\tilde{L}(x,y)$ do:
        i. $C(L_n) = Mean\{I_{blurred}(L_n)\}$
11. Else:
    a. $C(x,y) = \frac{1}{N} C(x,y)$
12. Return $C(x,y) * XDoG\{I(x,y)|\sigma_x, p, T(I)\}$

5.2 eXTENDED DIFFERENCE OF GAUSSIANS

Inputs:

- $I(x, y)$ – Input image (RGB or monochrome; floating-point).
- σ_x – Size of the XDoG Gaussian filter.
- p – Strength of the unsharp masking operator.
- $T(I)$ – Thresholding operator.

Outputs:

- $I_{XDoG}(x, y)$ – Output image (RGB or monochrome; floating-point)

Function $XDoG\{I(x, y)|\sigma_x, p, T(I)\}$:

1. If $I(x, y)$ is RGB:
    a. $I_g(x, y) = Rgb2Grey\{I(x, y)\}$
2. Else:
    a. $I_g(x, y) = Copy\{I(x, y)\}$
3. $G(x, y) = Gaussian\{I_g(x, y)|\sigma_x\}$
4. $DoG(x, y) = G(x, y) - G\{I_g(x, y)|1.6\sigma_x\}$
5. $U(x, y) = G(x, y) + p * DoG(x, y)$
6. $I_{XDoG}(x, y) = T(U(x, y))$
7. If $T(I)$ requires post-processing:
    a. $I_{XDoG}(x, y) = LineCleanup\{I_{XDoG}(x, y)\}$
8. Return $I_{XDoG}(x, y)$

5.3 SIMULATED ANTIALIASING

Inputs:

- $I(x,y)$ – Input image (binary; 1bpc).

Outputs:

- $I_{filtered}(x,y)$ – Output image (monochrome; floating-point)

Function $LineCleanup\{I(x,y)\}$:

1. $E_{horz}(x,y) = Convolve\{I(x,y)|[1 \quad 0 \quad -1]\}$
2. $E_{vert}(x,y) = Convolve\{I(x,y)|[1 \quad 0 \quad -1]^T\}$ (IMPORTANT: The kernel is transposed!)
3. $C(x,y) = E_{horz}(x,y) > 0 \wedge E_{ve}(x,y) > 0$
4. $B(x,y) = Convolve\left\{I(x,y)|\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}\right\}$
5. $I_{filtered}(x,y) = Copy\{I(x,y)\}$
6. $I_{filtered}\big(C(x,y)\big) = B\big(C(x,y)\big)$
7. Return $I_{filtered}(x,y)$