EE569 Homework #2

Xiang Gao
9216-3489-10
xianggao@usc.edu

October 13, 2013

1 Special Effect Image Filters

1.1 Motivation

Sometimes we'd like to add several special effect to images. In this problem, I will implement pencil sketch effect, background special effect as well as transition effect to image. In the end, I tried to acheive a method to get color pencil sketch effect. In pencil sketch effect, we focus on edges of image where we use several edge detectors and find which image is a best approximation of pencil sketch effect.

1.2 Approach

1.2.1 Color to Gray-Scale Conversion

In doing edge detection for color images, usually we should convert color image to gray-scale image first. However, if we just use any one of the RGB channels, we will lose a lot of information. Thus we employs YUV color space and pick Y channel as the corresponding gray-scale image.

YUV color space takes human perception into consideration, where Y is the luma (brightness) component and UV are chrominance (color) components. In this part we use Y component as the gray-scale image of original color image. The transfer function is as follows:

$$Y = 0.299R + 0.587G + 0.114B$$

1.2.2 Pencil Sketch using edge detectors

Since pencil sketch image mainly consists of edges, so to create pencil sketch effect to images, the first step is edge detection. Basicly, we use two basic edge detection method: 1)1st-order derivative gradient method and 2) 2nd-order derivative plus zero crossing method.

In first-order derivative method, we process the image with some first order derivative operations. Some typical 1st-order operators I used in experiments are as follows:[1]

Operator	Row gradient	Column gradient
Pixel difference	$\begin{bmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \end{bmatrix}$
Roberts	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$
Sobel	$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$

We can get gradient of each pixel with the operator above. Still we need a threshold to determine whether a pixel is an edge pixel or not. To get this threshold, I use the histogram of the gradient image and select 5 to 15 percent of pixels to be edge pixels.

As for 2nd-order derivative methods, I use the 2-D Laplacian operator: $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ to compute the 2nd order derivative of each pixel. We also need a threshold to get the three-level image, with which we can find the zero crossing point with the help of several patterns. Noting that because of the descrete effect, the value of edge in three-level image may not necessary be 0. Another 2nd-order method I used is "Difference of Gaussian"(DoG) method. In this method, we filter the image with 2 Gaussian lowpass filters with σ_1 and σ_2 respectively and subtract one result from the other. Typically DoG is an approximation to Laplacian of Gaussian(LoG) if $\sigma_2 = 1.6 * \sigma_1$[2].

1.2.3 Background Special Effect

We apply background effect to image by using this formula:

$$I_{out}(i,j,k) = I(i,j,k) + \alpha * G(i,j,k) + \beta$$

By adjusting α and β, we can change the weight of background image and foreground image.

1.2.4 Transition Special Effects

Sliding transition: By doing sliding transition, we just combine two images into one image considering the position.

Fade-in transition: By doing fade-in transition, we apply the transfer function:

$$I_{out}(i,j,k) = (0.5+\alpha) * I(i,j,k) + (0.5-\alpha) * G(i,j,k)$$

This formula adjust the weights of two input images by changing α and thus the two images will be blended togather.

1.2.5 Color Pencil Sketch

To implement color pencil sketch effect, I tried two methods. (1) The first one is described in the assignment. We apply pencil sketch effect algorithm to each of the RGB color channels and combine three sketch result to get a colored image. (2) The second method is to use pencil sketch result and blend this gray-level image with original color image[3]. I used this blending formula:

$$f(a,b) = \begin{cases} 2*a*b \ , if \ a < 0.5 \\ 1-2*(1-a)*(1-b) \end{cases}$$

where a is base layer and b is top layer. We can see from this formula, if base layer is bright(background), the result is also bright. If base layer is dark(edge), the result is a darker version of original color.

1.3 Results and Discussion

1.3.1 Color to Gray-Scale Conversion

(a) naruto_gray (b) harrypotter_gray (c) lena_noisy_gray

Figure 1: Gray-scale Images

1.3.2 Pencil Sketch using edge detectors

Choosing Threshold: A key problem in edge detection is how to choose the threshold of gradient so that we can seperate edge pixels from others. In this part, I use the histogram of the gradients to choose the threshold.

Since in 1st-order derivative method edge pixels have a larger gradient magnitude, the right side in the histogram will be chosen to be edge pixels. I choose the threshold based on the percentage of pixels (p) on the right side.

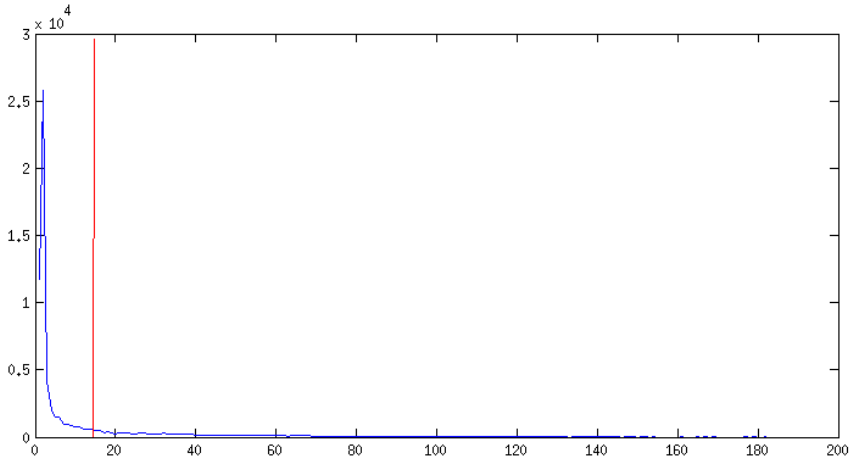


Figure 2: Threshold Selection of 1st-order

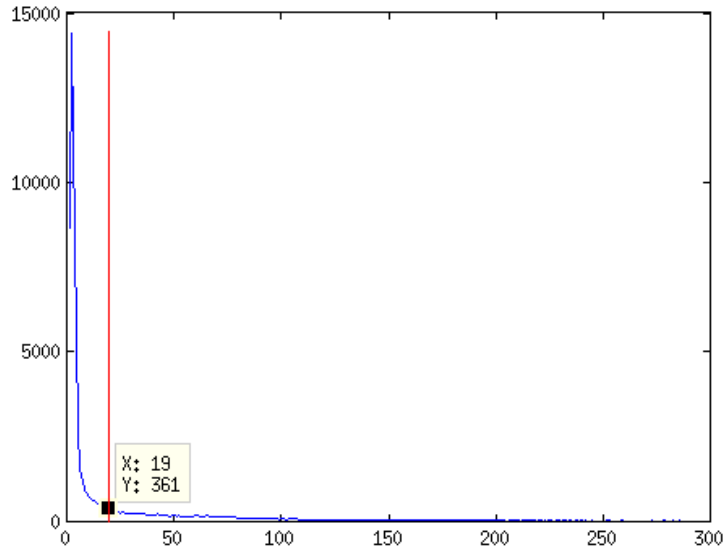


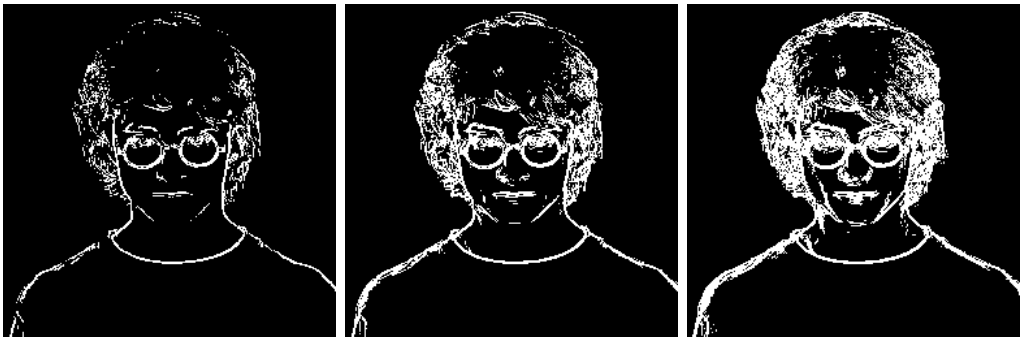
Figure 3: Threshold Selection of 2nd-order

In 2nd-order derivative (Laplacian) method, we have to choose a threshold to get the three-level image. Generally we have to choose a upper threshold and a lower threshold to seperate pixels to be -1,0,1. Since the histogram of 2nd-order gradient is usually symetric, I simply fold the left side to the right and find a positive threshold T. This this similar to the 1st-order's figure above. Then use -T and T as threshold to get three-level image.

Edge Detection: Implement derivative methods on "naruto.raw" image:

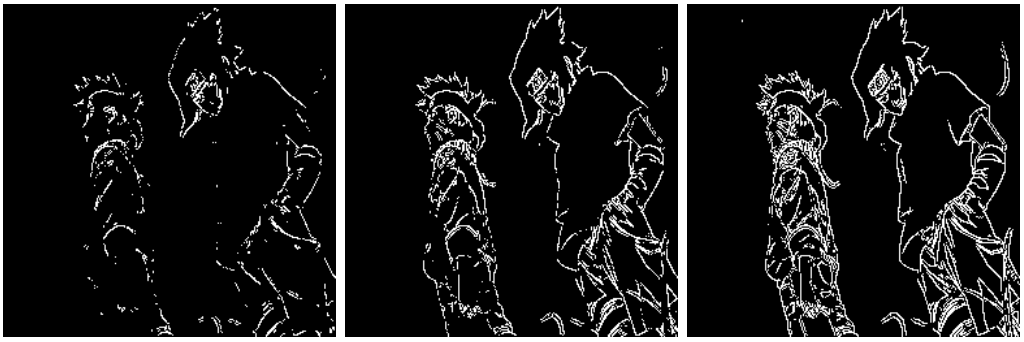
(a) p=5% (b) p=10% (c) p=15%

Figure 4: 1st-order Derivative Methods(using Pixel difference operator)



(a) p=5% (b) p=10% (c) p=15%

Figure 5: 1st-order Derivative Methods(using Pixel difference operator)



(a) p=5% (b) p=10% (c) p=15%

Figure 6: 2nd-order Derivative Methods(using Laplacian operator)

(a) p=5% (b) p=10% (c) p=15%

Figure 7: 2nd-order Derivative Methods(using Laplacian operator)

The more pixels we select as edges, the more details can be shown. Results of 2nd-order derivative method is output of zero-crossing pattern matching

of three-level image. We can see that result of 2nd-order edge detection is thinner than that of 1st-order. But this is actually not what we want as pencil sketch effect. Noting that the results looks quite clear because the "naruto.raw" image is almost free of noise.

Pencil Sketch Effect: Since we want to get a pencil-sketch-like output, we should notice some features of pencil sketch: (1)Pencil sketch is graylevel not just binary image and pencil color is usually black; (2) Pencil sketch could include not only edges of original image but also some details to make output look better. To acheive these 2 goals, I pay attention to gradient values. Gradient contains enough detials and can be converted to graylevel. I converted the gradient value to [0,255] and inverse it (larger gradient value has a smaller gray level which makes it darker in the image) to get a gradient image.

(a) 1st-order edges (b) 2nd-order edges (c) 1st-order gradient

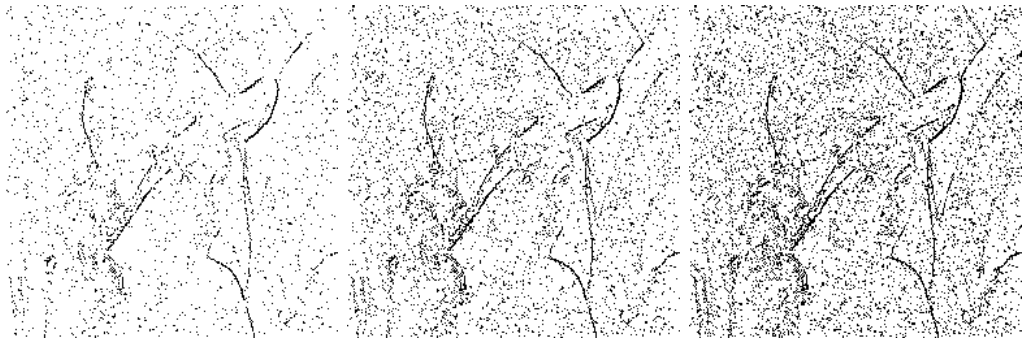
Figure 8: Pencil Sketch Effect Comparison: "naruto.raw"

(a) 1st-order edges (b) 2nd-order edges (c) 1st-order gradient

Figure 9: Pencil Sketch Effect Comparison: "harrypotter.raw"

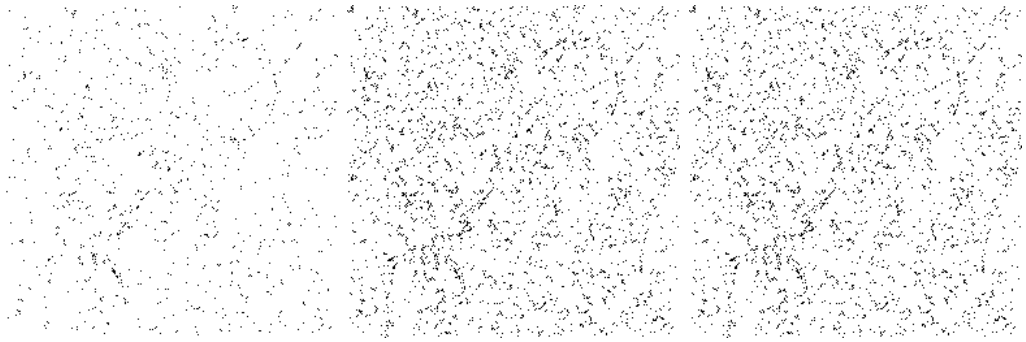
As the results show, I prefer to use gradient image as the pencil sketch effect, especially for the portrait image because it looks vivid and is rich in details.

Noisy Image: As for the noisy image, stories are totally different. If we implement these two kind of operator directly to a noisy image ("lena_noisy.raw"), we get the results below:



(a) p=5% (b) p=10% (c) p=15%

Figure 10: 1st-order Derivative Methods(using Pixel difference operator)



(a) p=5% (b) p=10% (c) p=15%

Figure 11: 2nd-order Derivative Methods(using Laplacian operator)

We can hardly see the actual edge especially with Laplacian operator if the image is noised. To deal with noised images, we implement "Difference of Gaussian" edge detection method[2]. With proper specifications (i.e $\sigma_2 = 1.6\sigma_1$), "DoG" is an approximation to "Laplacian of Gaussian" method. Hence I use DoG here to approximate LoG. With p=0.2 and window size = 7, I get the results below:

(a) $\sigma = 1.0$ (b) $\sigma = 2.0$ (c) $\sigma = 3.0$

Figure 12: Difference of Gaussian

DoG is more rubust to noise than first and second order derivative methods.

To use first and second order derivative methods, I use a Gaussian lowpass filter before edge detection. This gives better result:

(a) $\sigma = 0.6$ (b) $\sigma = 0.9$ (c) $\sigma = 1.2$

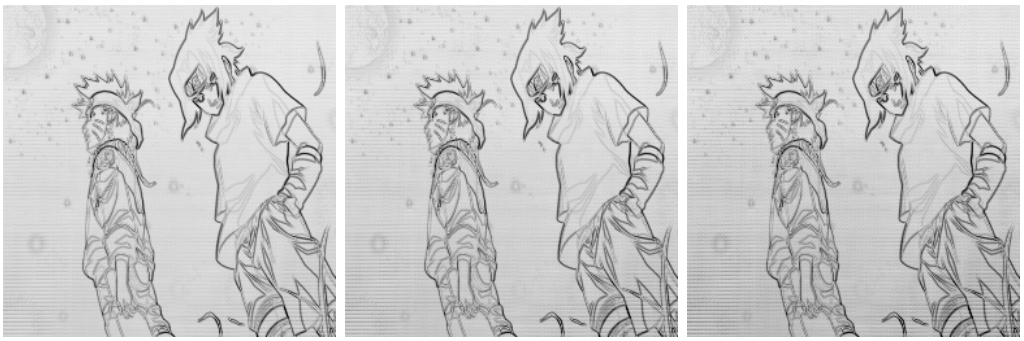
Figure 13: First order derivative with Gaussian Filter

The best pencil sketch images I get so far are as follows and I saved these files so that they can be used later.

(a) Naruto (b) Harrypotter (c) Lena

Figure 14: Pencil Sketch Images

1.3.3 Background Special Effect



(a) $\alpha = -0.55, \beta = 10$ (b) $\alpha = -0.75, \beta = 20$ (c) $\alpha = -0.95, \beta = 30$

(d) $\alpha = -0.55, \beta = 10$ (e) $\alpha = -0.75, \beta = 20$ (f) $\alpha = -0.95, \beta = 30$

(g) $\alpha = -0.55, \beta = 10$ (h) $\alpha = -0.75, \beta = 20$ (i) $\alpha = -0.95, \beta = 30$

Figure 15: Background Effect

1.3.4 Transition Special Effects

Figure 16: Sliding Transition Effect: Left to Right

Figure 17: Sliding Transition Effect: Left-top to Right-bottom

Figure 18: Fade-in Transition Effect

1.3.5 Color Pencil Sketch

Figure 19: Method 1

Method 1 uses RGB channels seperately and combines their results directly. Obviously, in some part of the image, there's edges in only one channel. This causes that the output color is of great difference with original color and makes image look weird.

Figure 20: Method 2: Naruto

Method 2 uses previous generated pencil sketch result, and blends this result with original color image. This will generally maintain original color

(a) Naruto (b) Harrypotter (c) Lena_noisy

Figure 21: Results using Method 2

2 Morphological Processing

2.1 Motivation

Using morphological processing, we can modify the spacial structure of an image without considering the color details of it. There are several useful morphological operations to help us explore the structures of image. In this task, I will implement the "Shrinking", "Thinning" and "Skeletonizing" operations with a 2-stage algorithm.

2.2 Approach

2.2.1 Shrinking

Since "star.raw" is a color image, the first step is to convert it to a binary image. I used Y channel as the intermediate gray image and then used a threshold to get final binary image. After shrinking operation, every connected object will be shrinked to one pixel assuming there will be no holes inside objects. As for the stars' sizes, I take the number of shrinking loops before it becomes one pixel as the size of that star.

2.2.2 Thinning

To count total number of digits, first we should convert this color image to a binary one. I also used Y channel as the corresponding gray image and take a threshold based on histogram of the gray image. After getting this binary image, every digit should be a connected object. Then I apply thinning operator to this binary image and every connected object is thinned.

To count the number of digits, I used an algorithm to count the number of connected object in the image. This algorithm takes a pixel and searchs around its neighborhood to find pixels belong to the same connected object.

2.2.3 Skeletonizing

Similarly to previous questions, I have to get the binary image first. There will be many small details after thresholding, so I apply a erosion operator to the binary image to remove those details. Then I use the skeletonizing operation to get the skeleton image.

2.2.4 Pacman Game

To count the number of point balls, I use shrinking method and keep track of each object's size. Since point balls are the smallest object, the number of objects that have smallest size will be the number of point balls.

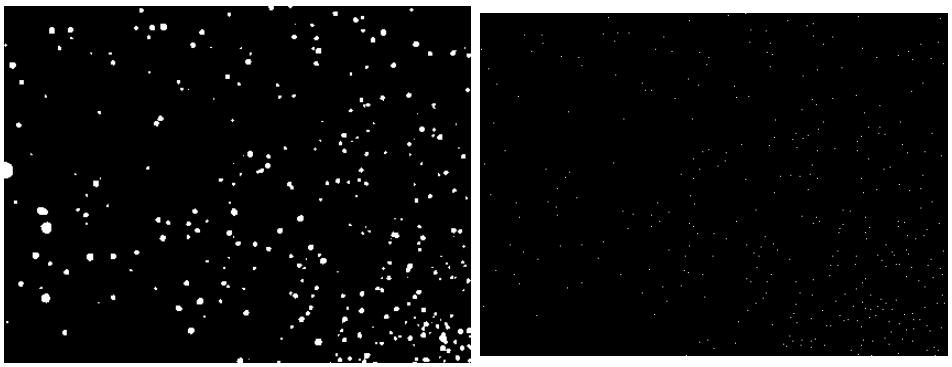
To count the number of walls, I use erosion operation and erode all small objects. Then with the help of "counting connected-object" algorithm that I mentioned previous, We can count the number of walls.

To count the number of turns, I used several patterns to find turns, which is described in result part.

2.3 Results and Discussion

2.3.1 Shrinking

With threshold = 30, I get a binary image, then shrink it to count the number of stars.



(a) Binary image: Left-top corner (b) After shrinking

Figure 22: Shrinking

From the left-top corner we can see all the points are shrinked to a single pixel after shrinking and we can simply count the number of pixel in the output image as the number of stars.

Total number of stars is 1796 and the histogram of sizes is drawn below:

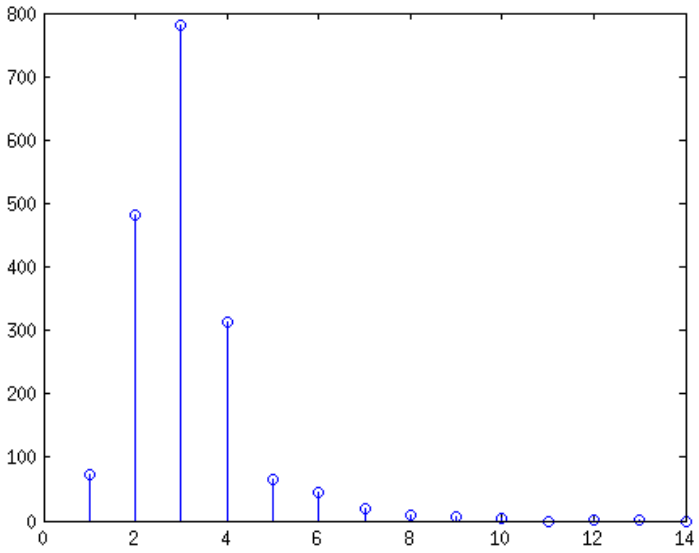


Figure 23: Histogram of stars' sizes

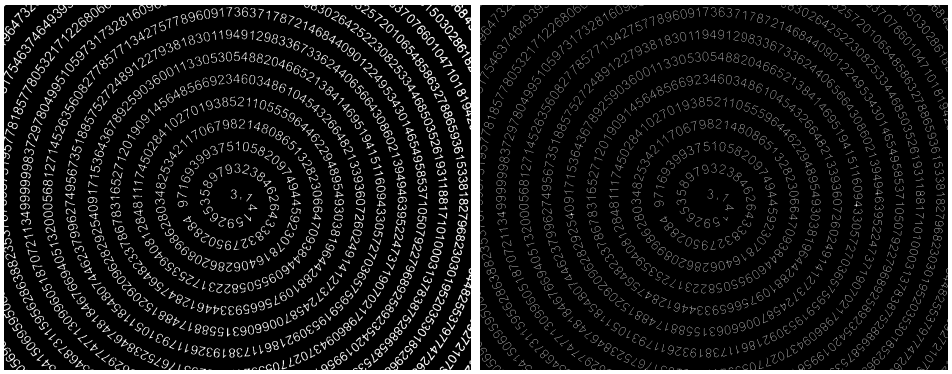
There's a problem with this method. Since we have to use a threshold, two stars that are very close to each other will be connected in the binary image and then be shrinked to one pixel. We can erode binary image first to seperate them while losing small stars.

```
Number of Stars: 1796
Star of size 1: 73
Star of size 2: 481
Star of size 3: 782
Star of size 4: 312
Star of size 5: 64
Star of size 6: 45
Star of size 7: 18
Star of size 8: 10
Star of size 9: 6
Star of size 10: 3
Star of size 11: 0
Star of size 12: 1
Star of size 13: 1
Star of size 14: 0
```

Figure 24: Counting Result

2.3.2 Thinning

With threshold = 125, I get a binary image, then thin it. The threshold is carefully chosen to deduce the background color as well as to maitain the connection within digits.



(a) Binary image (b) After thinning

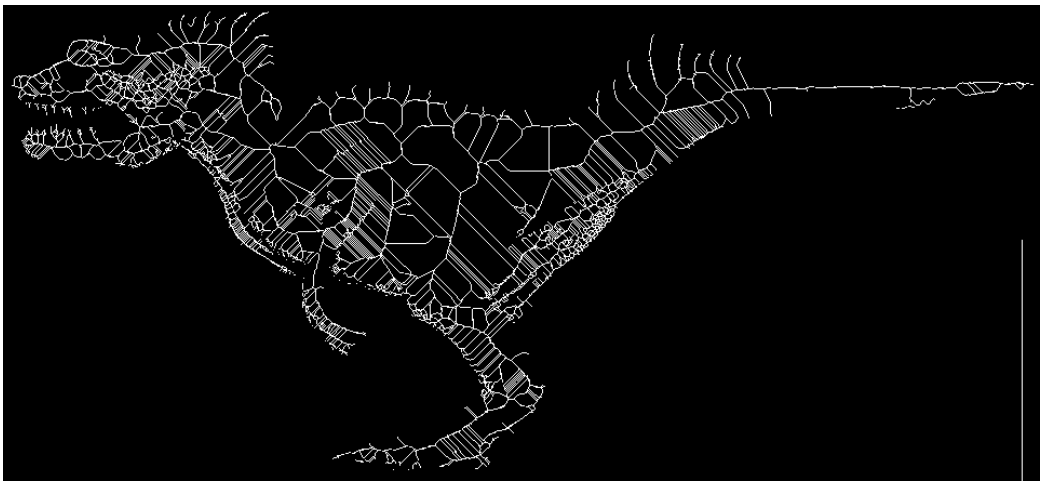
Figure 25: Thinning

Here's trick in counting the number of digits. Since there are many partial digits on the boundary and we don't want to count small partial objects as digits, I used a threshold to only count larger objects. Threshold I used here is 10 pixels and I get the total number of digits is 1054.

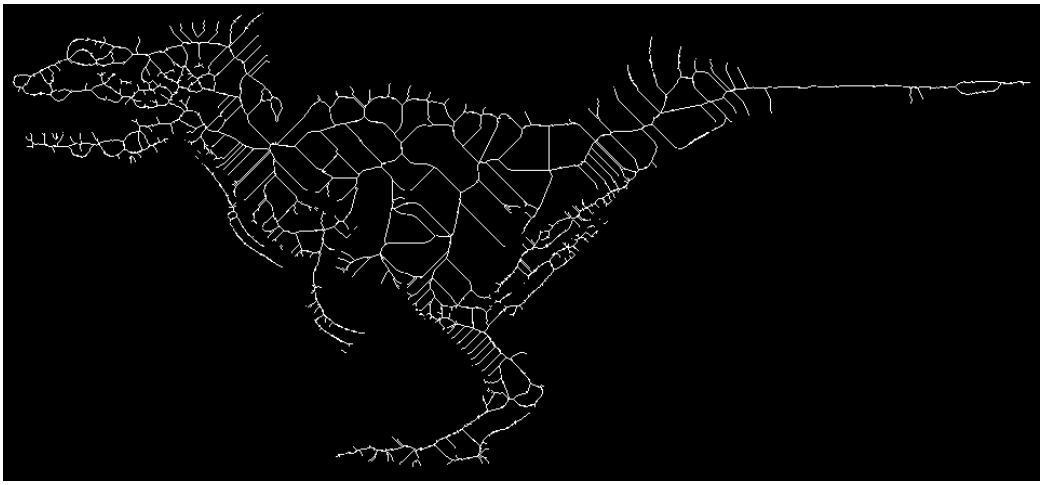
```
Number of Digits: 1054
```

Figure 26: Counting Result

2.3.3 Skeletonizing



(a) Skeletonizing directly



(b) Skeletonizing after erosion

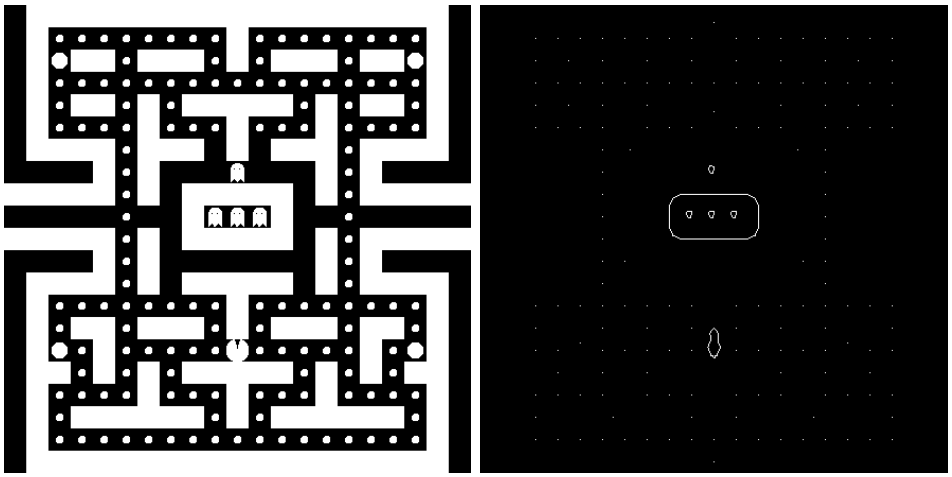
Figure 27: Skeletonizing

We can see using erosion operator, we have removed a lot of unuseful details (the undesired line on the right is also removed) and get a better skeletonizing result.

2.3.4 Pacman Game

To do the following tasks, the first step is to convert the original image to a binary one. The threshold used in this step is easy to choose.

Count point-balls: To count the number of point-balls in the image, I have to shrink all objects. After shrinking, all objects are converted to points or circles. To seperate point-balls from other objects, I use the size of them to determine because of the fact that point-balls are the smallest objects in the image.



(a) Binary image (b) After Shrinking

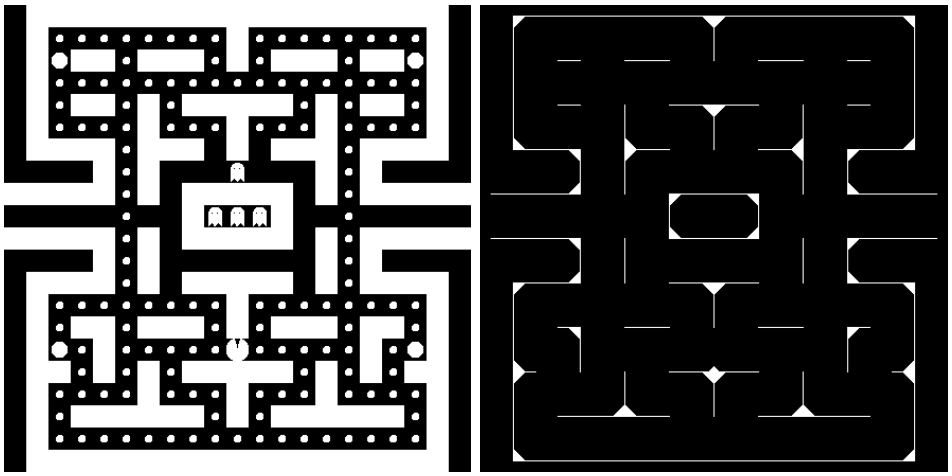
```
Object of size 1: 0
Object of size 2: 0
Object of size 3: 0
Object of size 4: 146
Object of size 5: 0
Object of size 6: 0
Object of size 7: 0
Object of size 8: 4
Object of size 9: 0
Object of size 10: 0
Object of size 11: 0
Object of size 12: 0
Object of size 13: 0
Object of size 14: 0
Object of size 15: 0
Object of size 16: 0
Object of size 17: 0
Object of size 18: 0
Object of size 19: 0
Object of size 20: 0
Object of size 21: 0
Object of size 22: 4
```

(c) Result

Figure 28: Count point-balls

The number of smallest objects is 146 which is the number of point-balls.

Count walls: To count the number of walls, I use erosion operator to erode all small balls as well as pacman and ghosts. Then I implement an algorithms to count number of connected object in the image. Thus I can get the number of walls in the image.



(a) Binary image (b) After Erosion

```
Connected objects number(size >= 1): 22
```

(c) Result

Figure 29: Count walls

Count turns: In this task, I count all the turns in the image. I use 4 5x5 patterns to determine turns (which is not the same with corners). Patterns I used is:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

After using these patterns to check hits in the binary image, I get the number of turns:

```
Number of turns: 114
```

Figure 30: Count turns

By doing this, I also count 4 turns where the pacman will never reach.

3 Digital Half-toning

3.1 Motivation

Halftoning is usually used in binary printing where we can only use black and white color. We want to convert graylevel images to binary images without losing many details. With halftoning, original graylevel image is represented with one color dots and this can be blended by human eyes.[5] In this part, I will implement dithering halftoning method as well as error diffusion methods. In the end, I will also implement an algorithm that convert halftone image into original graylevel image.

3.2 Approach

3.2.1 Dithering

Fixed thresholding: We choose a fixed value as threshold and set all pixels which below this threshold to be black and above it to be white. Transfer function is as follows:

$$G(i,j) = \begin{cases} 0, \ if \ 0 \leq F(i,j) < T \\ 255, \ if \ T \leq F(i,j) < 256 \end{cases}$$

where T is the fixed threshold. Clearly, it can be expected that we will loose many details with this method.

Random thresholding: We use a random threshold for each pixel in this method. Transfer function is:

$$G(i,j) = \begin{cases} 0, \ if \ F(i,j) < rand(i,j) \\ 255, \ if \ F(i,j) \geq rand(i,j) \end{cases}$$

where rand(i,j) is a random threshold which follows a certain distribution funtion. In this task, I will try uniform distribution and normal distribution generators.

Dithering Matrix: We use a matrix to determine the threshold in each position within a block. Matices are defined as $\begin{bmatrix} 4 * I_n(x,y) + 1 & 4 * I_n(x,y) + 2 \\ 4 * I_n(x,y) + 3 & 4 * I_n(x,y) \end{bmatrix}$

and threshold for each position $T(x,y) = \frac{I(x,y)+0.5}{N^2} * 255$. Transfer funtion:

$$G(i,j) = \begin{cases} 0, \ if \ F(i,j) \leq T(i \ mod \ N, j \ mod \ N) \\ 255, otherwise \end{cases}$$

where N is the size of dithering matrix. In this task, I will implement 2x2 and 4x4 dithering matices.

3.2.2 Error Diffusion

Since methods wiht dithering matrix will introduce new patterns into halftoning result which is not wanted, we will use error diffusion method. In error diffusion method we calculate the error from halftoning of each pixel and diffuse it to its following neighbors. In this task, I will implement 3 different error diffusion matrices and they are as follows:
(1)Floyd-Steinberg:

$$\frac{1}{16} * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 7 \\ 3 & 5 & 1 \end{bmatrix}$$

(2)Jarvis, Judice and Ninke Error diffusion (JJN):

$$\frac{1}{48} * \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 5 \\ 3 & 5 & 7 & 5 & 3 \\ 1 & 3 & 5 & 3 & 1 \end{bmatrix}$$

(3)Stucki:

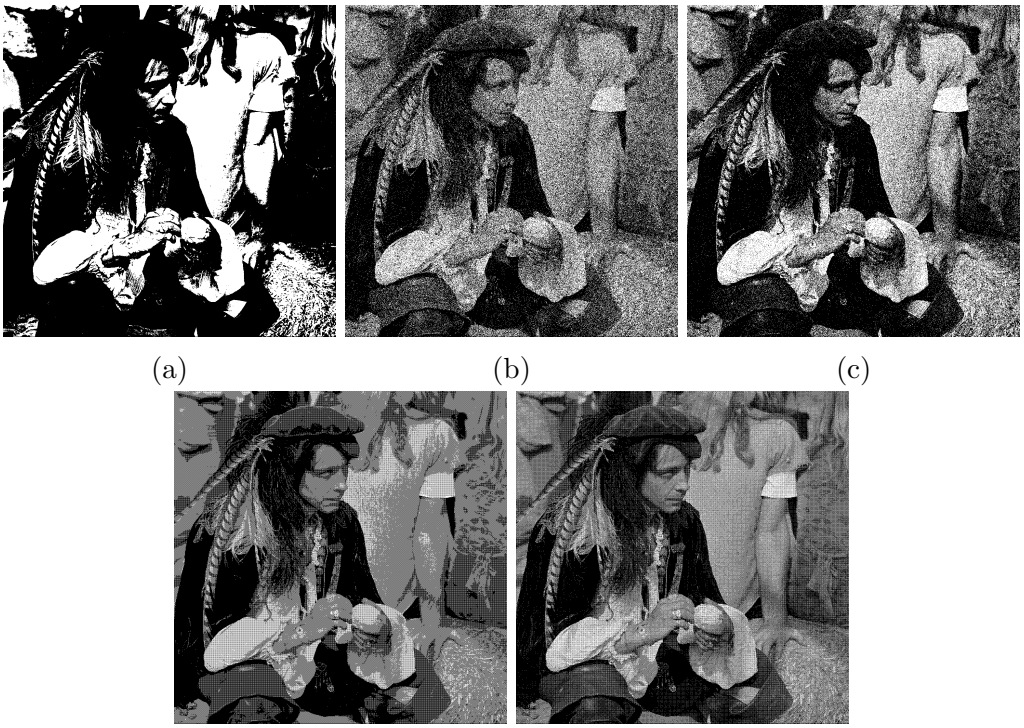
$$\frac{1}{42} * \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 4 \\ 2 & 4 & 8 & 4 & 2 \\ 1 & 2 & 4 & 2 & 1 \end{bmatrix}$$

3.2.3 Inverse Half-toning

In this part I implement a inverse halftoning algorithm from [4]. This cascade algorithm uses run-length method to get a graylevel at the first stage and employs a adaptive spatial low-pass filter in the second stage to smooth the result and finally remove some impulses by checking the variance of its neighborhood.

3.3 Results and Discussion

3.3.1 Dithering



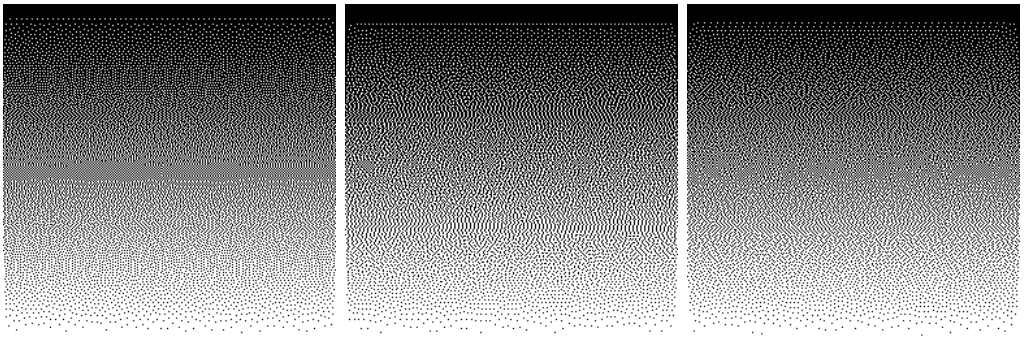
(a) (b) (c)

(d) (e)

Figure 31: Dithering: (a)Fixed thresholding, (b)Random thresholding, Uniform, (c)Random thresholding, Gaussian($\mu = 127, \sigma = 50$), (d)Dithering Matrix: 2x2, (e)Dithering Matrix: 4x4

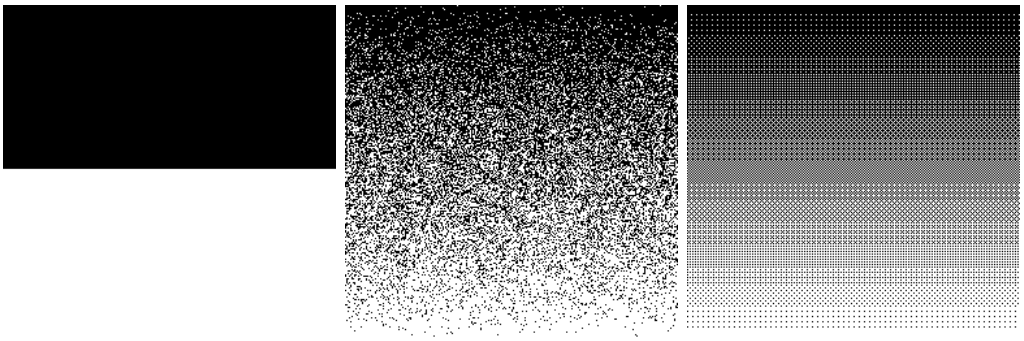
From the results we can see that the result of "fixed thresholding" is not satisfying since it looses a lot of information. Comparing uniform and Gaussian generator, I prefer Gaussian generator because the result looks much sharper. Using dithering matrix, the results looks smoother but invite texture that is not from original image.

3.3.2 Error Diffusion



(a) Floyd-Steinberg (b) JJN (c) Stucki

Figure 32: Error Diffusion

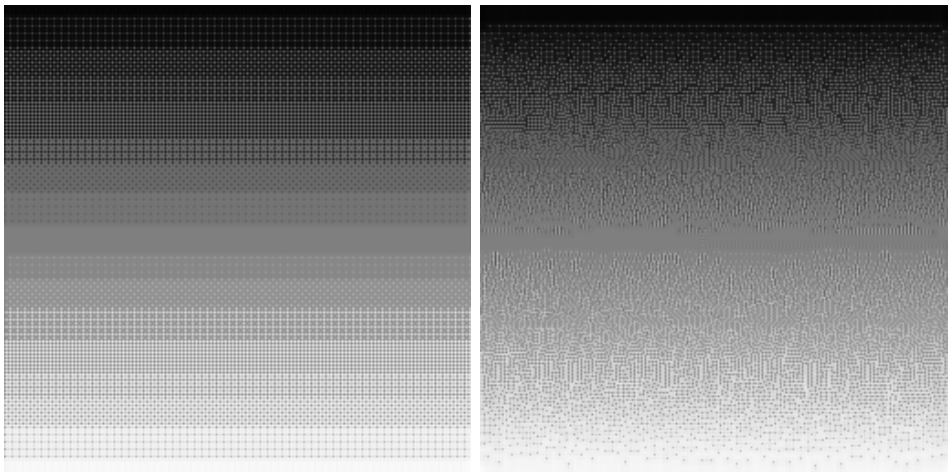


(a) Fixed thresholding (b) Random thresholding (c) Dithering Matrix: 4x4

Figure 33: Results of Dithering

Comparing results of dithering method and error diffusion matrix, we can see that error diffusion matrix method gives a smoother result without patterns.

3.3.3 Inverse Half-toning



(a) Inverse halftoning of Dithering Matrix (b) Inverse halftoning of Floyd-Steinberg

Figure 34: Inverse Half-toning

(a) Inverse halftoning of Dithering Matrix (b) Inverse halftoning of Floyd-Steinberg

Figure 35: Inverse Half-toning

As the author mentioned in this paper[4], the cascade inverse halftoning algorithm works particularly well for error diffusion techniques. Because halftoning image from dithering matrix will has patterns that is rich of edges

and the cascade algorithms is designed to reserve edges, the result is not as good as error diffusion techniques.

References

[1] William K. Pratt, "Digital Image Processing", 4th Edition, John Wiley & Sons Inc., 2007.

[2] [Online]: `http://en.wikipedia.org/wiki/Difference_of_Gaussians`

[3] [Online]: `http://en.wikipedia.org/wiki/Blend_modes#Overlay`

[4] C. M. Miceli and K. J. Parker, "Inverse halftoning" Journal of Electronic Imaging, vol. 1, no. 2, pp. 143151, 1992.

[5] [Online]: `http://en.wikipedia.org/wiki/Halftone`