IMAGE PROCESSING

In QT, OpenGL, & C++

By Xuewei Fan
CSC - Senior Design Capstone
Professor George Wolberg
Grove School of Engineering
at City College of New York
December 19, 2016

Table of Contents

Table of Contents

Introduction

Project Overview

This is an image-processing project we implemented for our Senior Design class at the City College of New York. Digital image processing is computer manipulation of pictures or images that have been converted into numeric form. The finished application lets you perform various image filters. In particular this application can be used to edit an image by applying point operations and neighborhood operations on that image. Point operations include Threshold, Clip, Quantization, Gamma Correction, Contrast Enhancement, Histogram Stretch, and Histogram Match functionalities. Neighborhood operations include Blur, Sharpen, Median, Convolve, Error Diffusion, Correlation functionalities. In this project, we only focused on three neighborhood operations: Blur, Convolve, and Correlation. The sample window in Figure 1 shows correlation filter applied on the maid image.

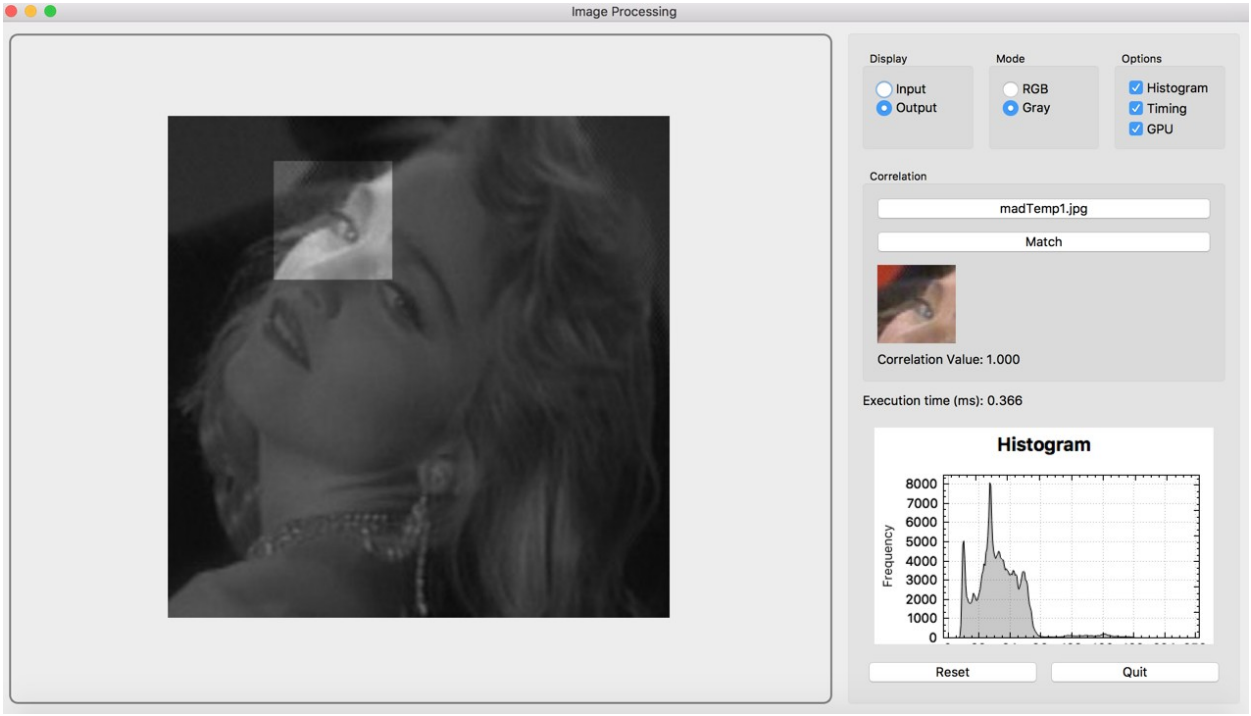


Fig. 1. Correlation

Technologies

The goal of this project is to create an image processing application by using OpenGL, QT and C++. OpenGl (Open Graphics Library) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. It is typically used to interact with a graphics processing unit (GPU) to achieve hardware-accelerated rendering. It means using GPU based OpenGL to render image is more faster than using CPU. We used GLSL in our implementation. GLSL (GLslang) is short for the official OpenGL Shading Language. GLSL is a C/C++ similar high level programming language for several parts of the graphic card. With GLSL we coded short programs, called shaders, which are executed on the GPU to render our images.

Program Design & Display

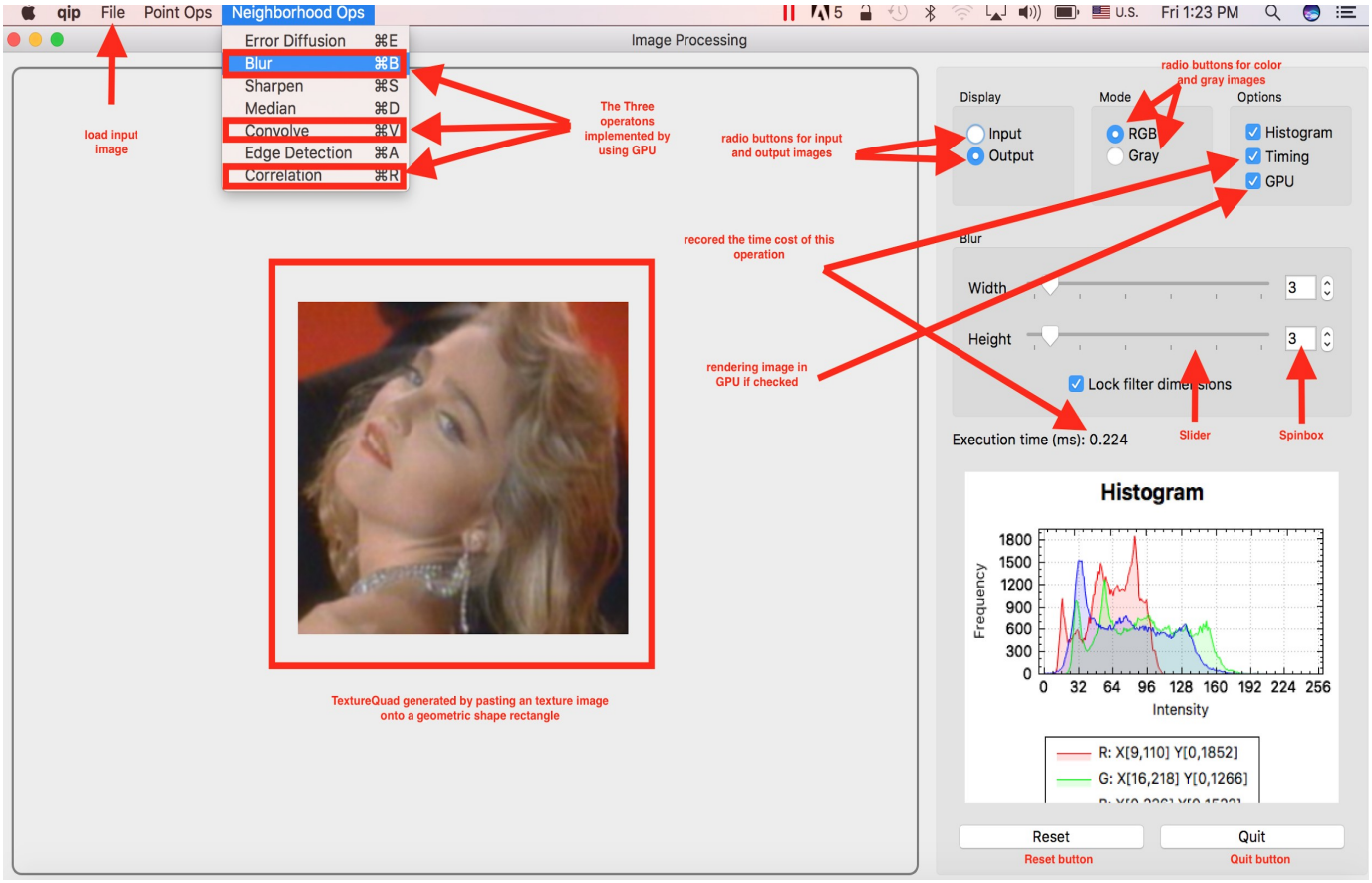


Fig. 2. User Interface with QtWidget and GLWidget

The rest of the image processing operations have their own .h and .cpp files where their classes, respective functions and layouts are defined. The right part of the above window is a control panel that changes for each different image filter. The top left groupbox contains 2 radio buttons. The 2 radio buttons let you select between displaying input or output image. The middle groupbox contains two radio buttons, which let you choose between displaying a grey scale or color (RGB) image. The top right groupbox contains 3 checkboxes: Histogram, Timing, and GPU. The Histogram checkbox when checked displays the histogram. The histogram shows the frequency of 0-255 pixel values in the image. The timing checkbox when checked displays the time it took to perform the operation in milliseconds. The GPU checkbox when checked uses the

Graphic Processing Unit of our computer to perform the image filters with OpenGL. If this checkbox is unchecked then the operations are performed in CPU. The empty groupbox in the above picture is a container for the image filters. In the following pages the respective widgets for each filter will be displayed in this groupbox. Instead of using the Qlabel for displaying input and output images in previous project we are using GLWidget in this project to display input and output images. The left part in the above window is the GLwidget where the input and output image is displayed. The left part remains consistent for all of the different image filters.

Point Operations

Threshold

Definition

The threshold filter separates and assigns all input pixel values from the input image to 0s or 255s and display them on the output image. As the figure 3 shows below, it basically checks each pixel value on the input image, if the input pixel value is below m, then assign its corresponding output value to black (0). If input pixel value is above m, then assign its corresponding output value to white (255). Figure4 is the Lut table that is set in advance and used to check each input pixel value and assign the computed value to its corresponding output pixel.

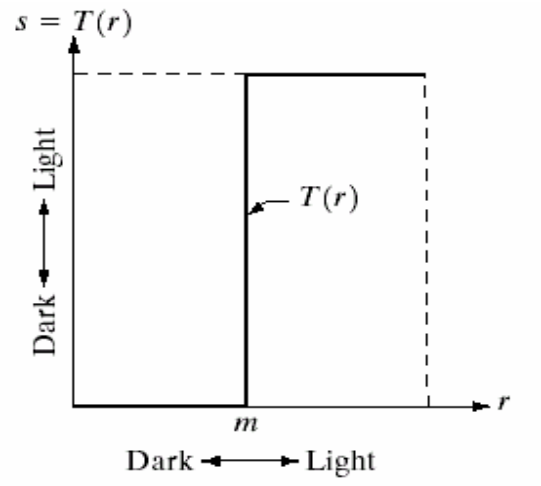


figure 3

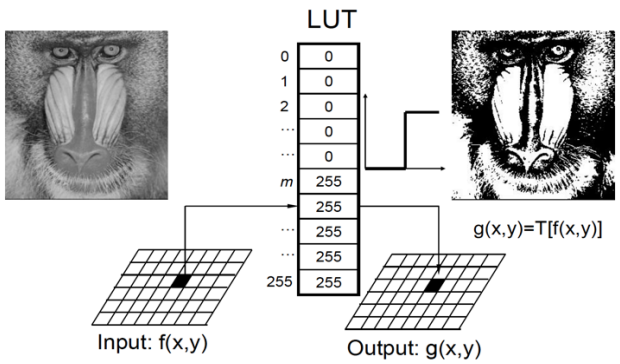


figure 4

Demo

Fig. 5. Original Input Image

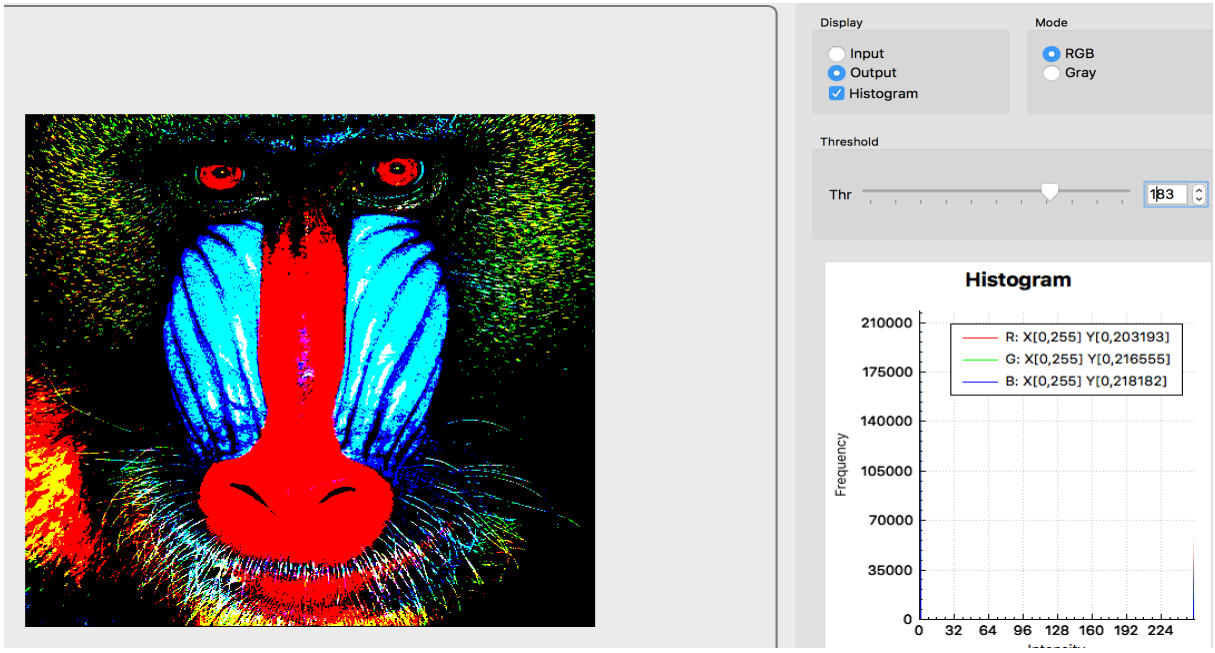


Fig. 6. The output image when the threshold value is set to 183

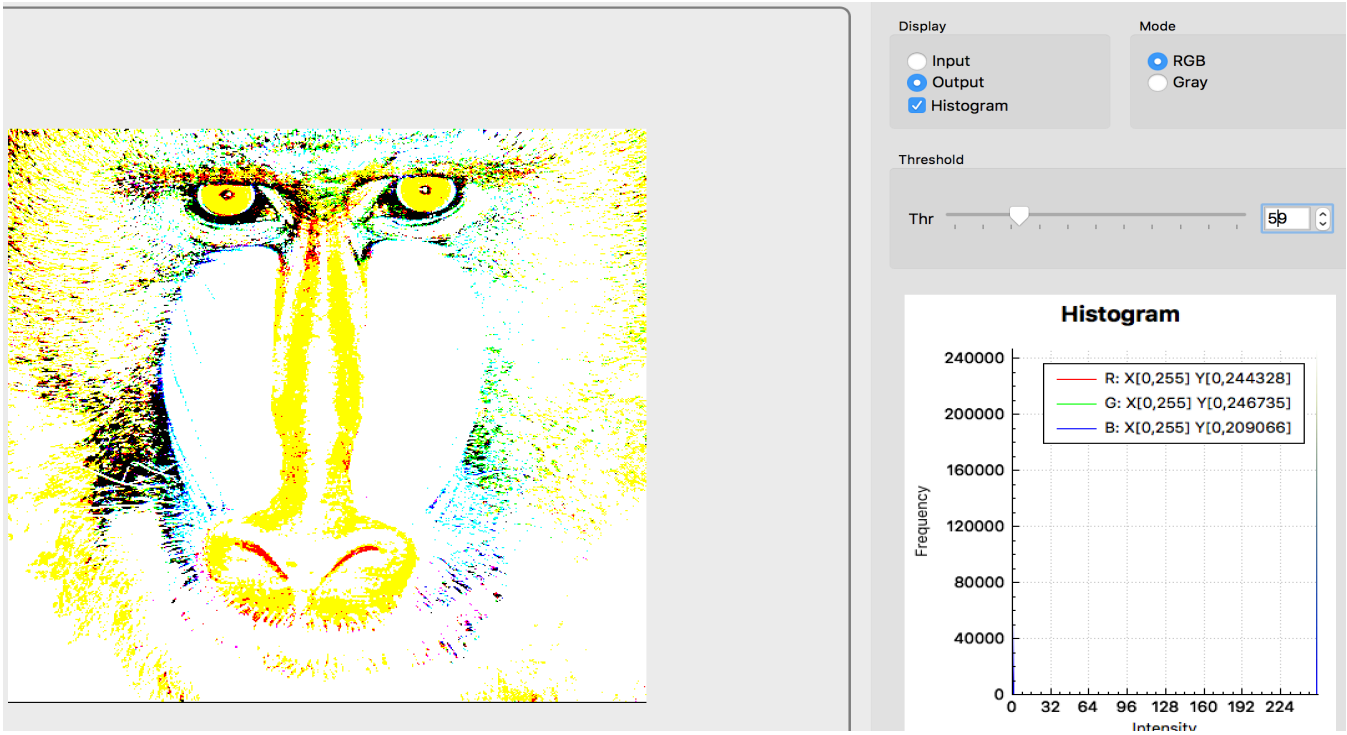
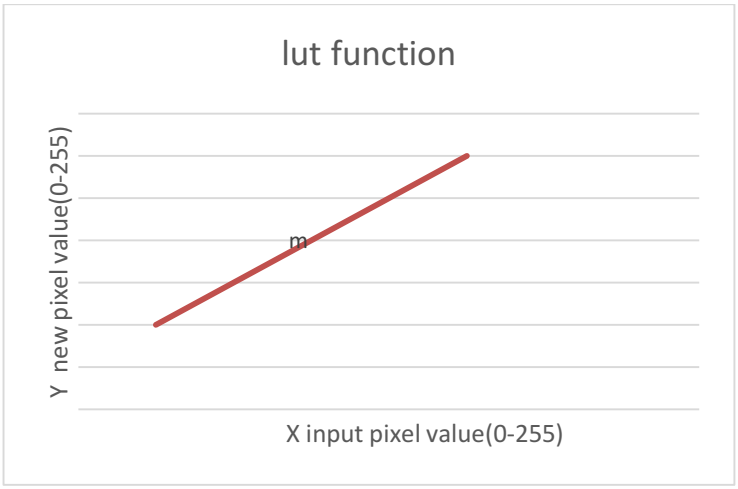


Fig. 7. The output image when the threshold value is set to 59

Contrast

Definition

Contrast is the difference between maximum and minimum pixel intensity. If the output pixel values cover a broad range of gray scale (0-255) and the distribution of pixels is not too far from uniform, with very few vertical lines being much higher than the others, then the image is high-contrast. If the output pixel values cover a narrow range of gray scale (0-255) and spread over only the center of the gray scale, then the image is low-contrast. Variable contrast is the slope of the lut function. See the chart below, if the slope is large, the difference in value is large between the pixels above m and below m, there will be more input pixel values clipped to 0 and 255 as well, it means the contrast is high. If the slope is flat, it means there is no much difference among all the pixels, so the contrast is low, there will be less input pixel values clipped to 0 and 255.



Demo

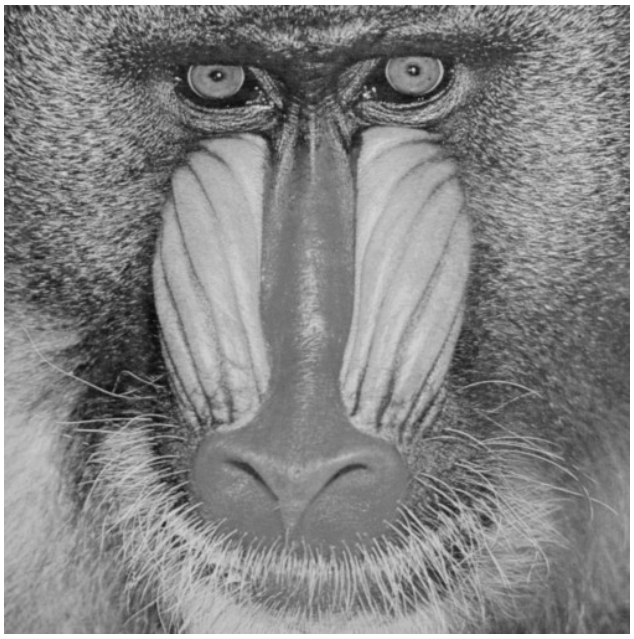
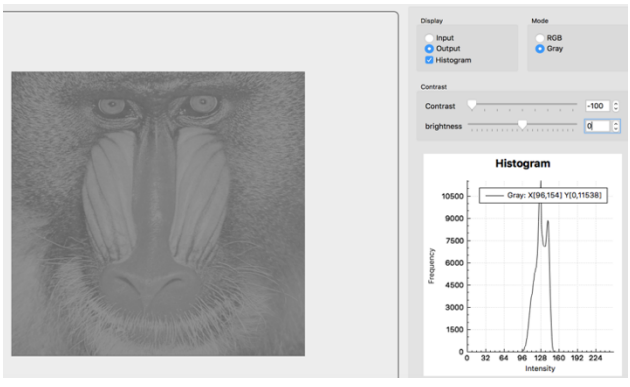
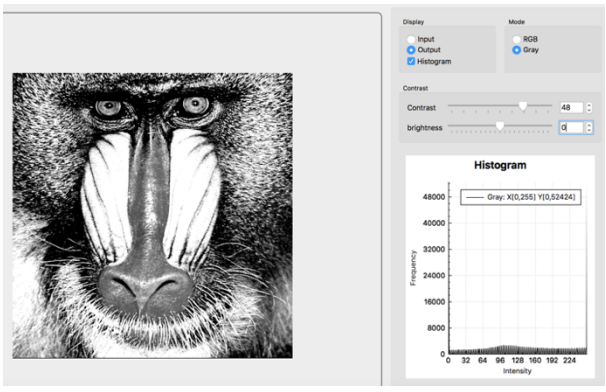


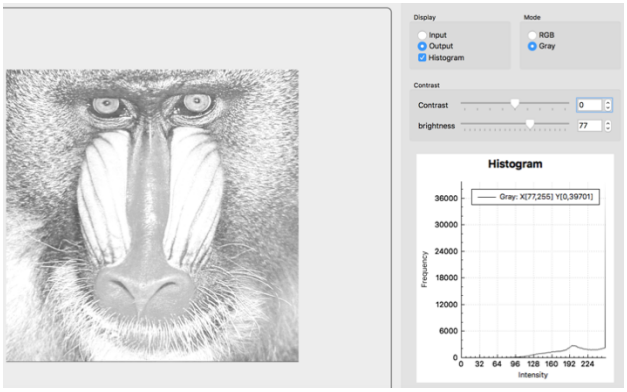
Fig. 8. Original Input Image



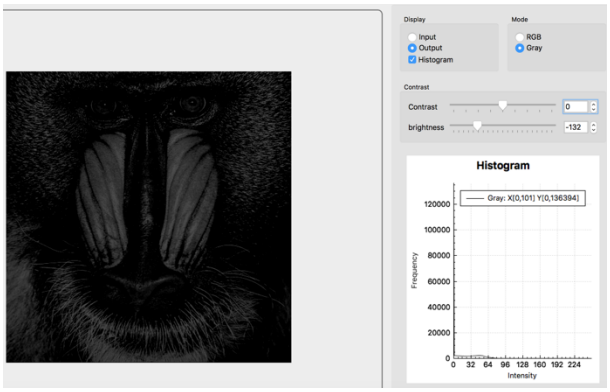
(a) Low contrast



(b) High contrast



(c) High brightness



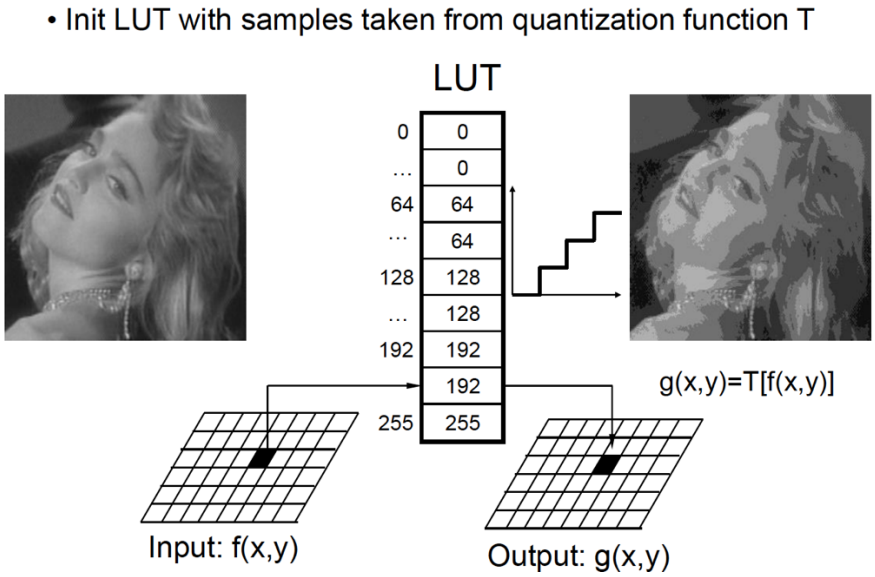
(d) Low Brightness

Fig. 9

Quantization

Definition

Quantization is a lossy compression technique achieved by compressing a range of values to a single quantum value. Basically it is the mapping of real numbers on a finite set, a many to one mapping. In this implementation, we will map the input pixels into different intervals based on the quantization level the user set. For example, if the quantization level = 2, then scale = 256/2 = 128, we map all pixels with the value less than 128 to the first interval in which we reset all the pixel values to scale*1. Similarly, we map all pixels with the value greater than 128 to the second interval in which we reset all the pixel values to scale*2. The functionality of Dither is to reduce the quantization error by uniformly distributed the white noise (dither signal) to the input image prior the quantization.



(Wolberg: Image processing note)

Demo

Fig. 10. Original Input Image

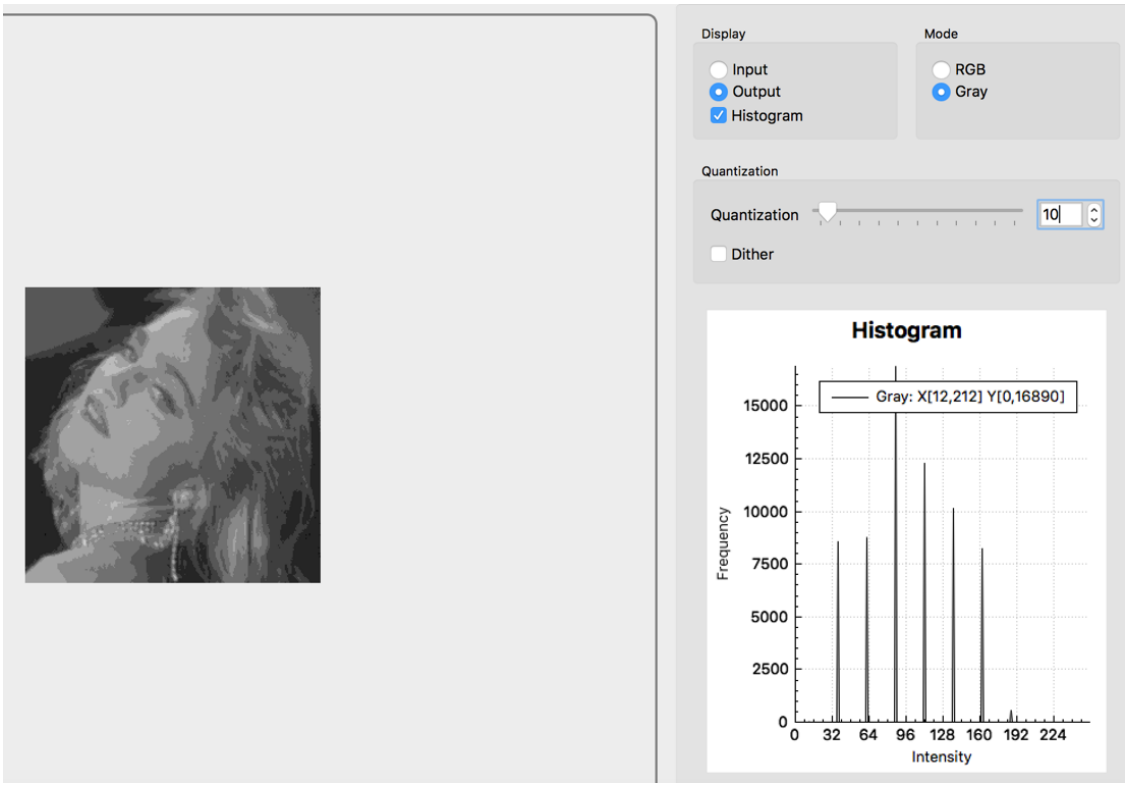


Fig. 11. Output image with quantization level 10

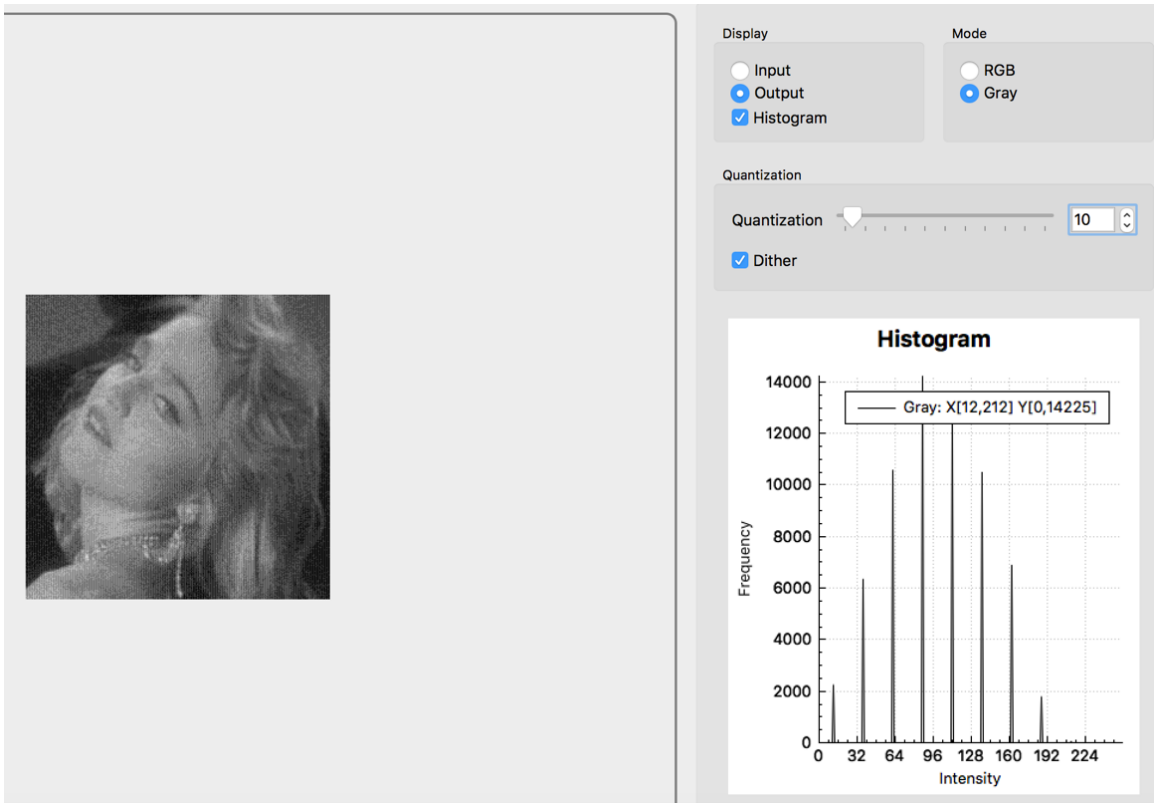
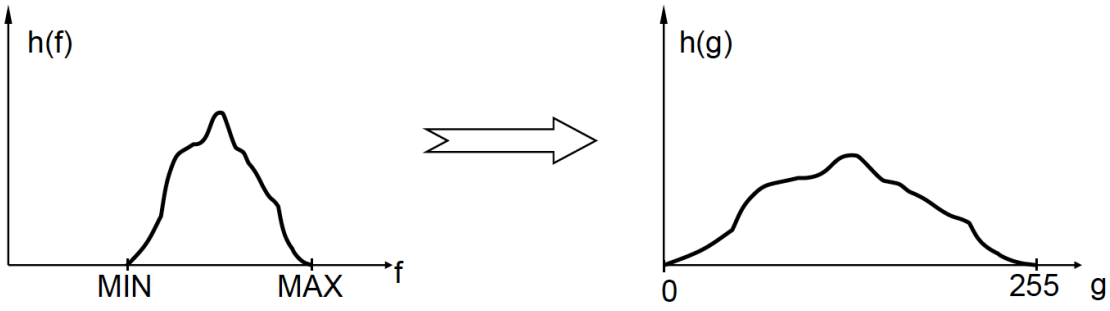


Fig. 12. Output image with quantization level 10 and with dither signal applied on it

Histogram Stretching

Definition

Histogram stretching a process that changes the range of pixel intensity values. It attempts to improve the contrast in an image by `stretching' the range of intensity values it contains to span a desired range of values. Histogram matching is used for image enhancement (enhance contrast). As the graph shows below, where f denotes the value of each pixel intensity. For each f in an image, we will calculate this formula. User can specify MIN and MAX. If the input pixel value f is less than MIN, clip it to zero.



3) Rescale to [0,255] range ← → **1) Slide histogram down to 0**

$$g = \frac{255(f - MIN)}{MAX - MIN}$$

→ **2) Normalize histogram to [0,1] range**

(Wolberg: Image Processing Course Notes)

Demo

Fig. 13. Original Input Image

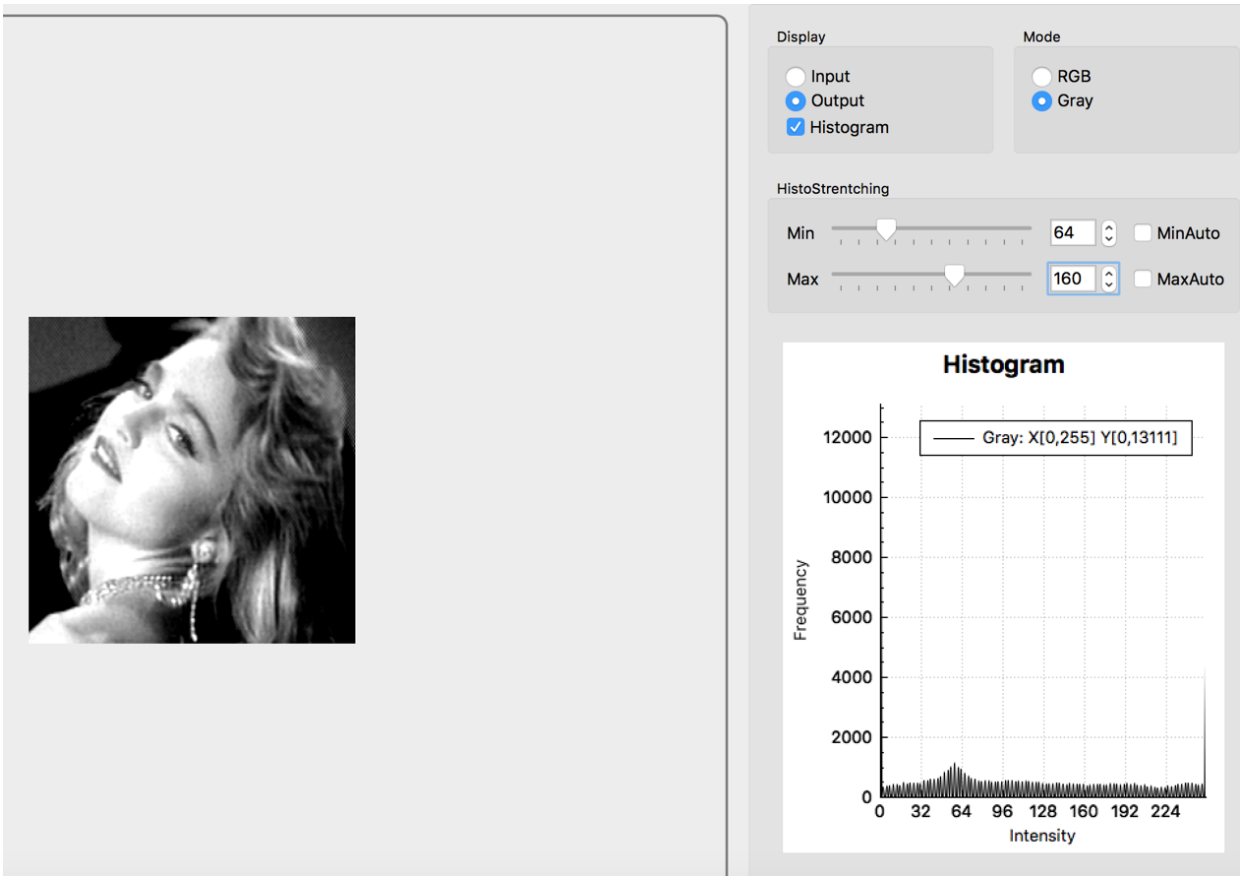


Fig. 14. The image after set the Min to 64 and Max to 160, it stretches the histogram that has the range from 64 to 160 to the range from 0 to 255.

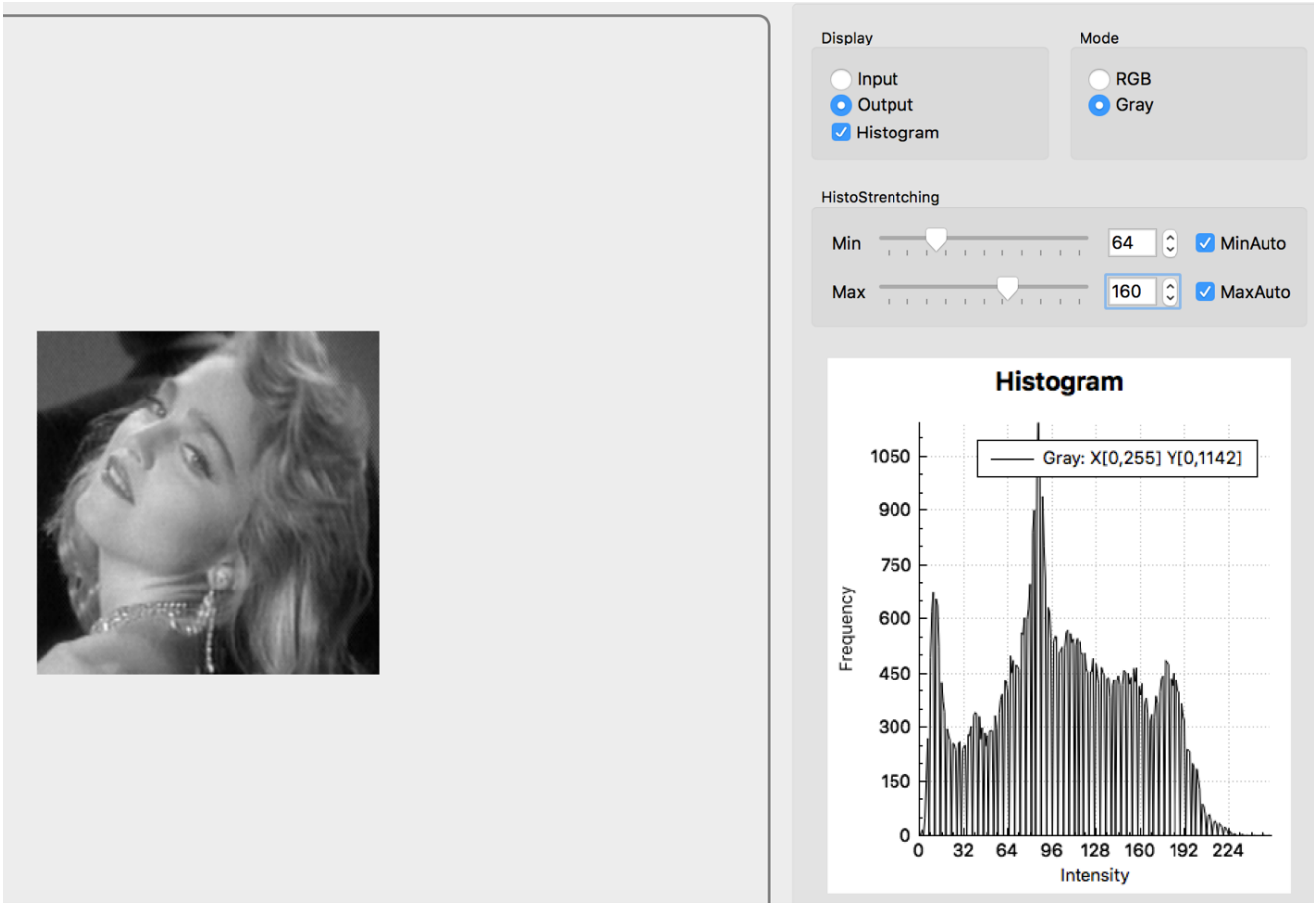


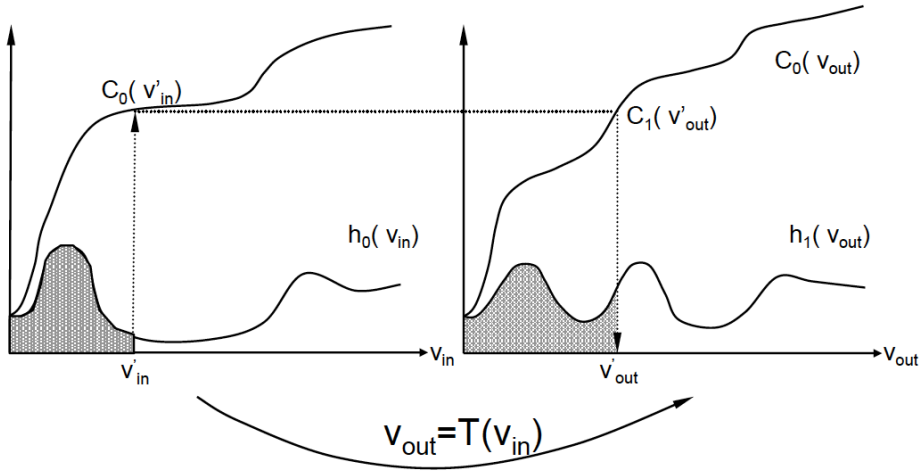
Fig. 15. The output image and its histogram after the MinAuto and MaxAuto is checked. We can see the histogram is similar to the original input image. That's because after the MinAuto and MaxAuto are checked, MAX and MIN values passed in the function are the maximum value of the pixel and the minimum value of the pixel of the input image respectively. The histogram slides down to 0 and up to 255.

Histogram Matching

Definition

histogram matching is the transformation of an image so that its histogram matches a

specified histogram. Histogram matching is used for image enhancement.



In the figure above, *h()* refers to the histogram, and *c()* refers to its cumulative histogram. Function *c() is a monotonically increasing function defined as:*

$$c(v) = \int_0^v h(u)du$$

(Wolberg: Image Processing Course Notes)

the origin histogram and matched histogram need meet the equation:

$$\int_0^{v_{out}} h_1(u)du = \int_0^{v_{in}} h_0(u)du$$

This can be restated as:

$$c_1(v_{out}) = c_0(v_{in})$$

Where *c1(vout)* = # pixels <= *vout*
c0(vin) = # pixels < =*vin*

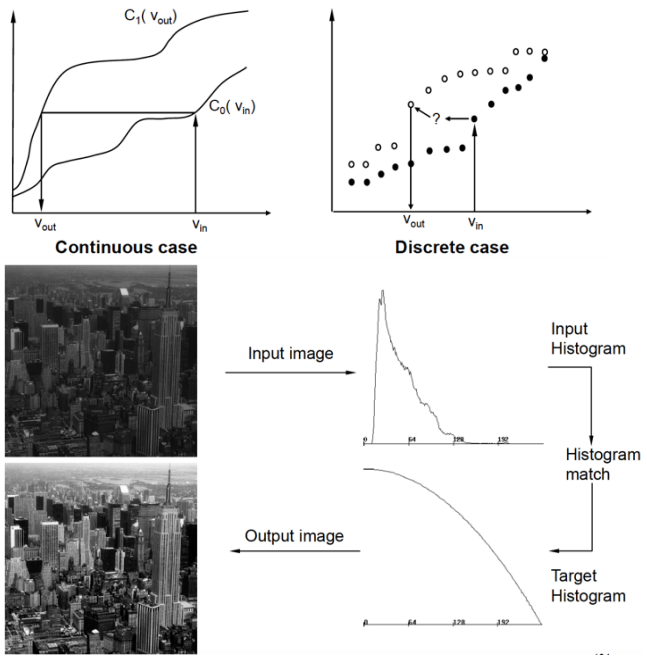
And we can get the basic equation for histogram matching:

$$v_{out} = c_1^{-1}(c_0(v_{in}))$$

The set of input pixel values is a discrete set, it is impossible to match all histogram pairs. So we can't get inverse for c1 in $v_{out} = c_1^{-1}(c_0(v_{in}))$ because of discrete domain.

Solution: choose Vout for which c1(Vout) is closest to c0(Vin), such that

|c1(Vout)-c0(Vin)|is a minimum.



(Wolberg: Image Processing Course Notes)

Demo

Fig. 16. Original Input Image

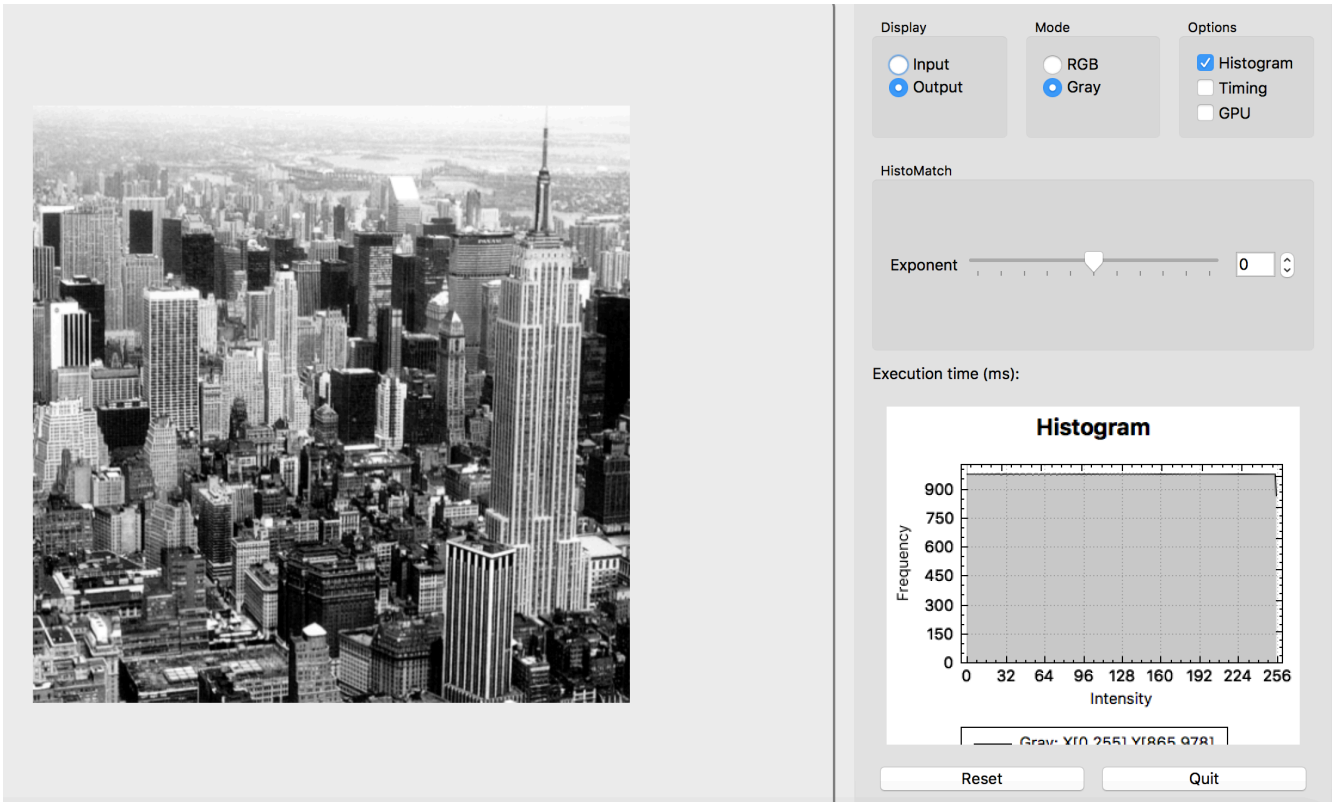


Fig. 17. This is the output image and its histogram when exponent n = 0

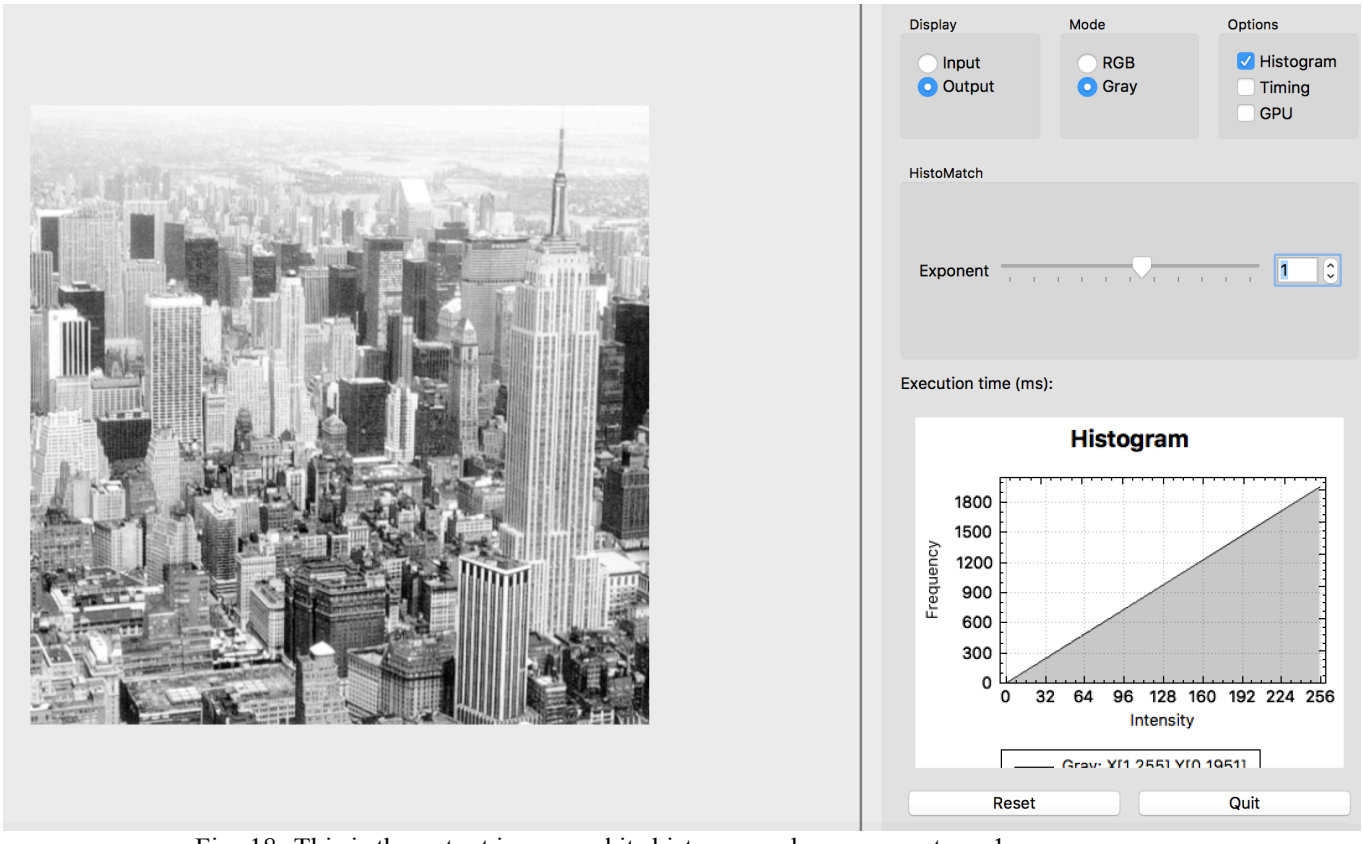


Fig. 18. This is the output image and its histogram when exponent n = 1

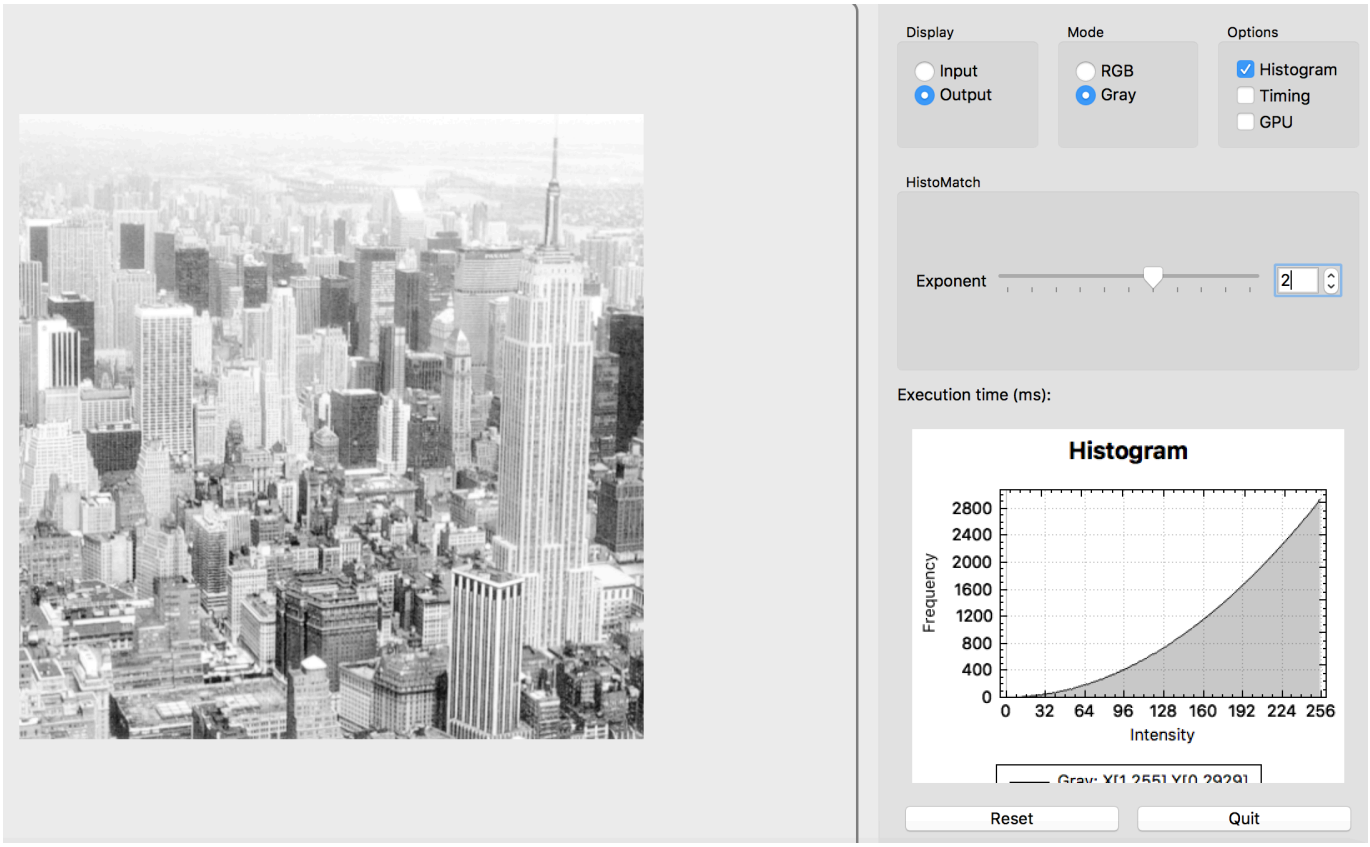


Fig. 19. This is the output image and its histogram when exponent n = 2

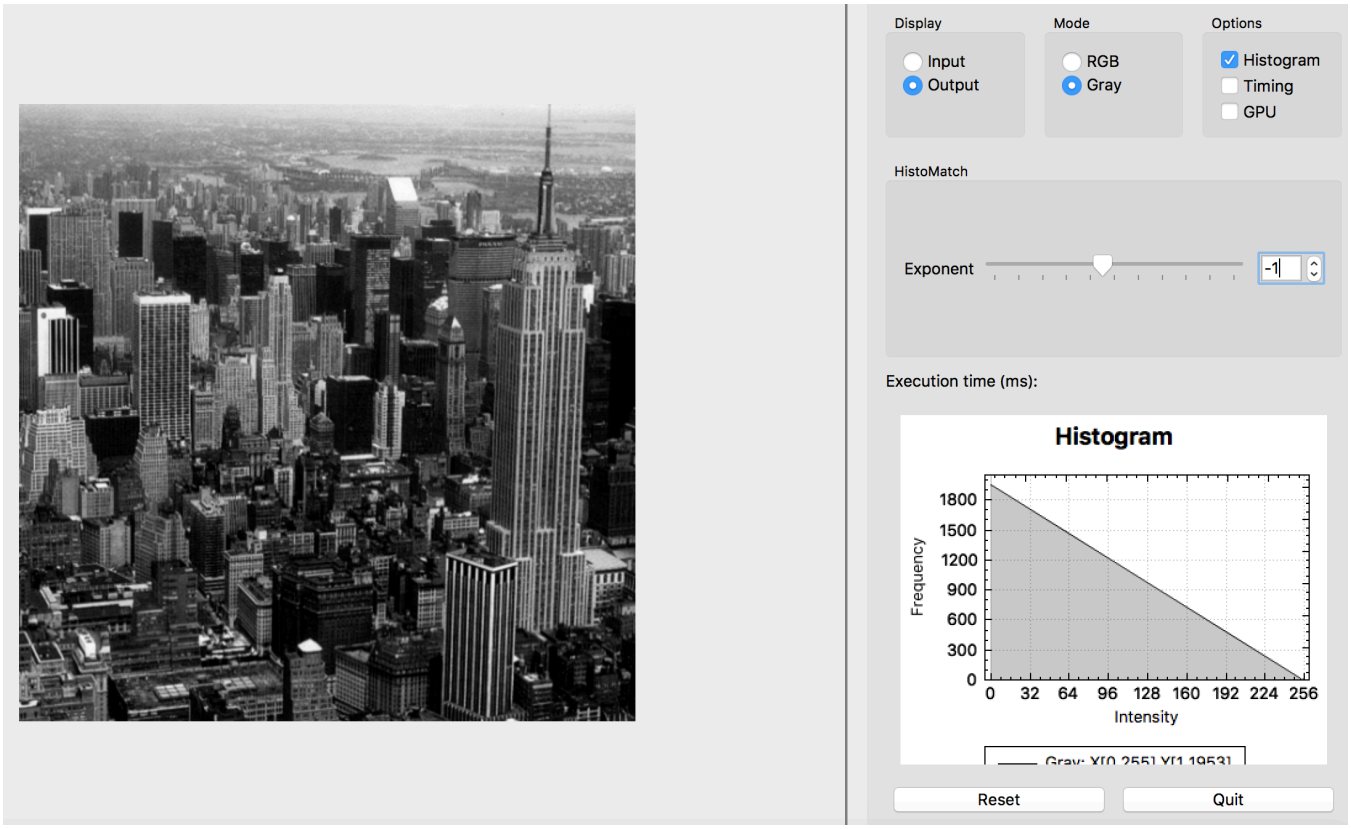


Fig. 20. This is the output image and its histogram when exponent n = -1

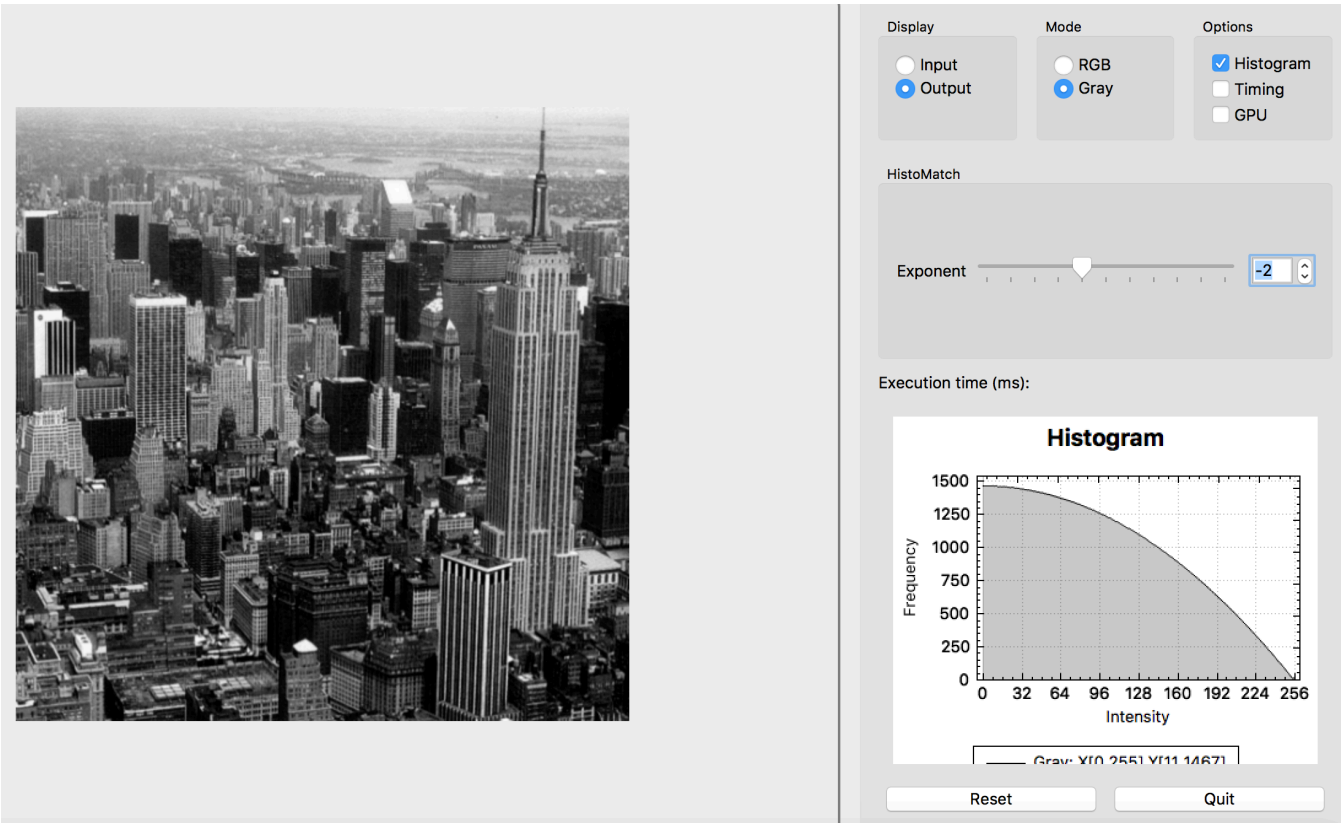


Fig. 21. This is the output image and its histogram when exponent n = -2

Neighborhood Operations with OpenGL

In neighborhood operations output pixels are a function of several input pixels. Transformation T is applied to few input pixels to calculate the value for each output pixel. The transformations are implemented with buffers. A collection of input pixels is stored in the buffer at a time.

Transformation T is applied to the values in the buffer, and the resulting value is stored in the output pixel.

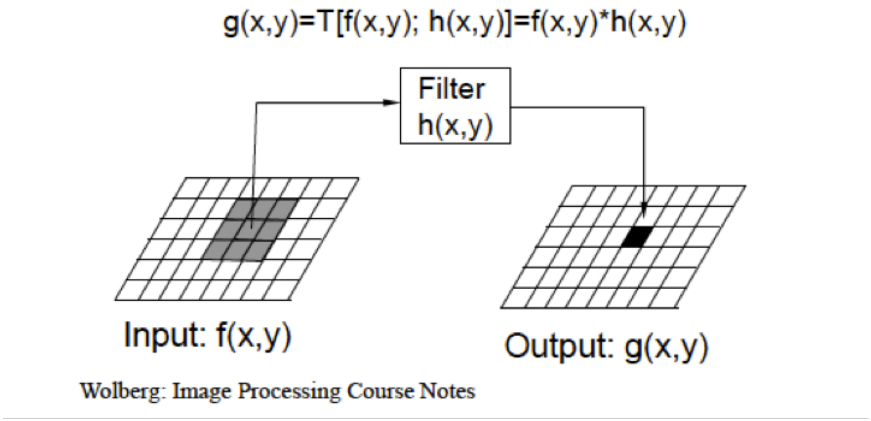


Fig. 22

The main task of this project is to implement three neighborhood operations: Blur, Convolve and Correlation in a faster way with the GPU rendering. The programing language used in this project is C++, GLSL. The framework used here is Qt. The image rendering library used here is OpenGL. In this project, Instructor provided the basic templates and libraries for us to complete all the implementations. Basically, we created a GUI interface in QT and coded the shader program for each of the functionality. Each interface let users load an image and perform editing on that image by choosing different options on the interface. The main workflow of OpenGL is shown in the picture below.

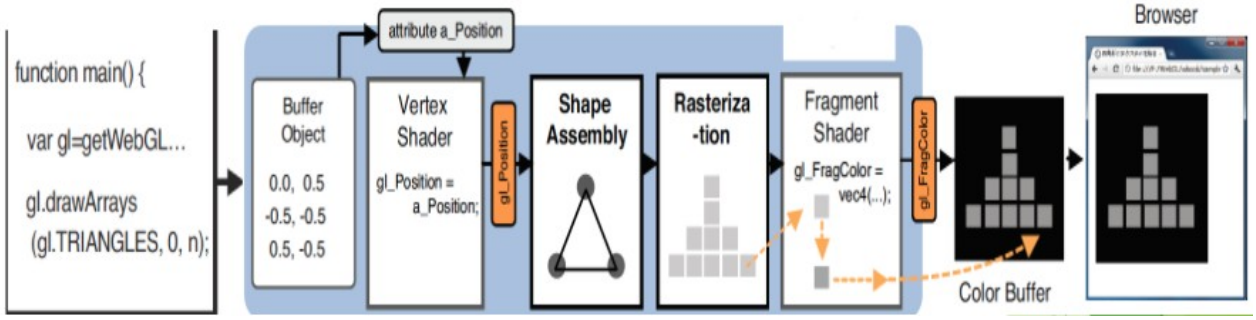


Fig. 23. Assembly and rasterization between a vertex shader and a fragment shader(Matsuda & Rodger)

In our project the vertex shader manipulates the attributes of the vertices of the image and sends 4 vertices to the shape assembly. Shape assembly makes a primitive from the vertices and passes the geometric shape to rasterizer. The rasterizer converts the geometric shape into fragments (pixels) and passes those fragments to the fragment shader. The fragment shader

processes each pixel and based on the filter takes care of how the output pixels look.

Blur

Definition

In blurring we reduce the edge content of an image, which makes the transition from one color to the other color very smooth. We apply blur with a box filter that has dimensions xsz * ysz.

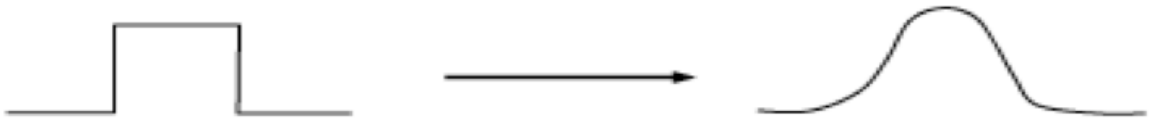


Fig. 24

Demo

Fig. 25. Original Input Image

The following images show blur applied in both direction and blur applied in one direction only.

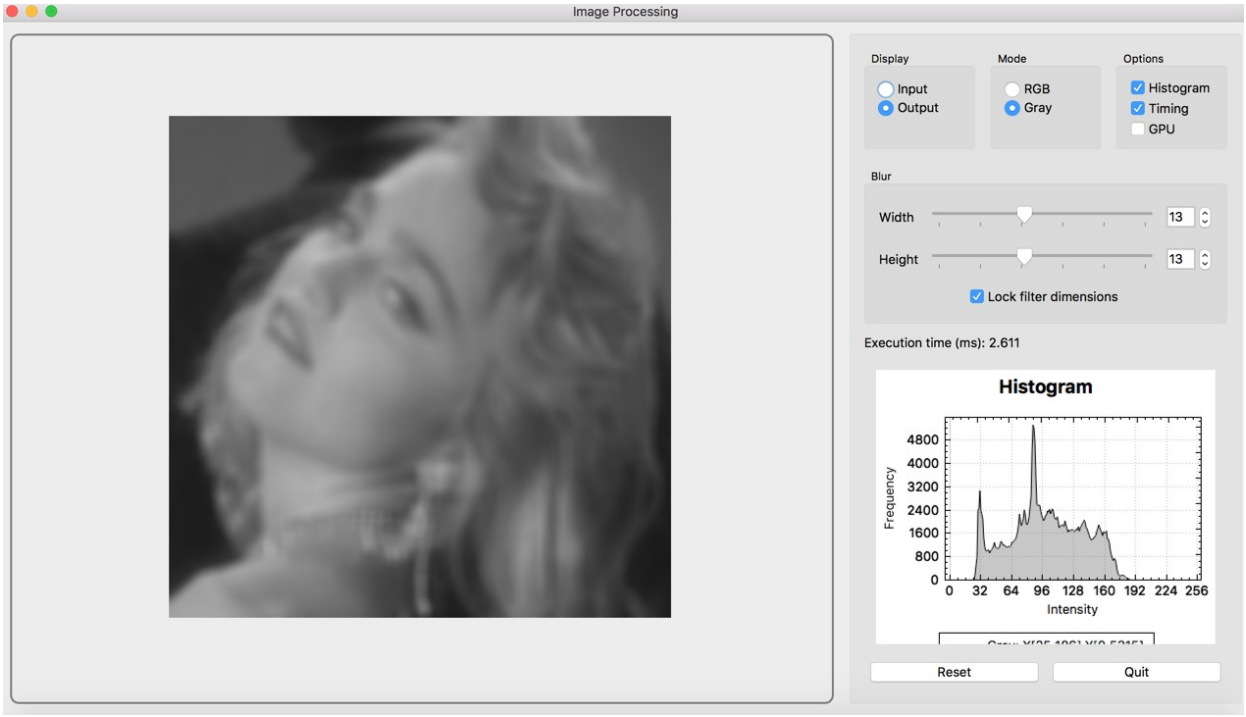


Fig. 26. Image after applied a 13x13 blur kernel on the original image (CPU)

Execution time (ms): 2.611 At the middle right of the panel we can see the execution time is 2.611 milliseconds in CPU. This time is the average time after we run 100 times of this operation.

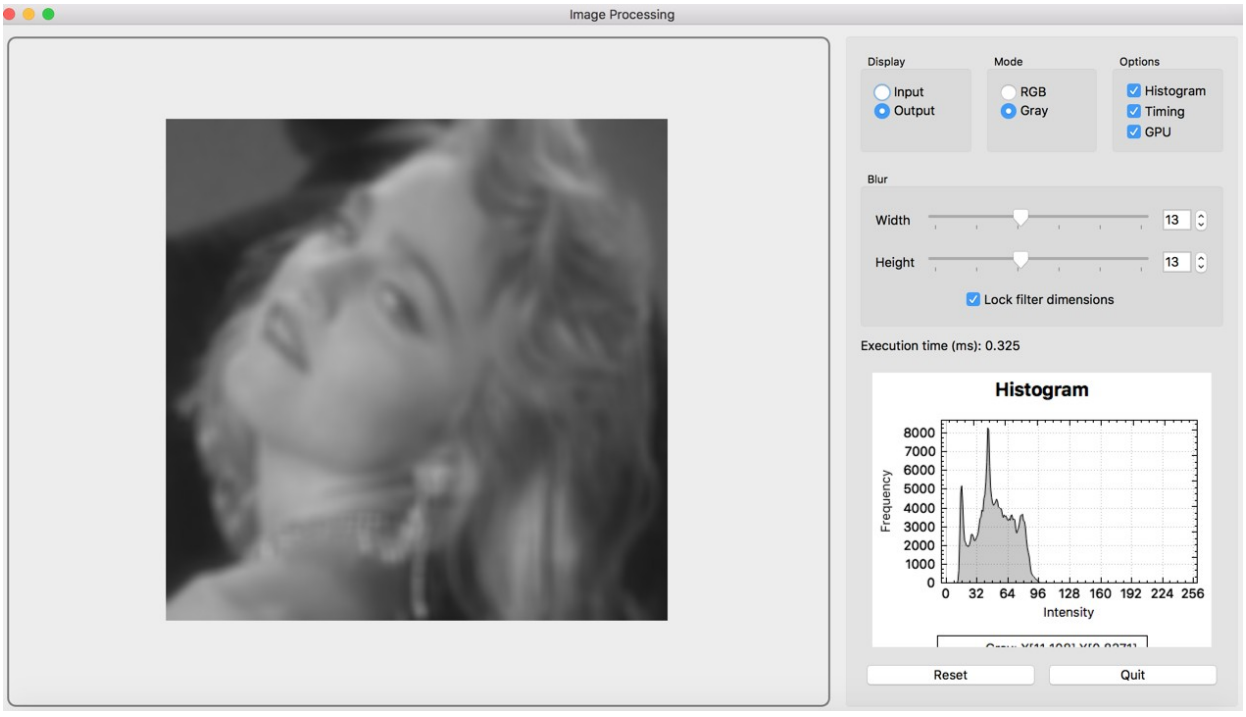


Fig. 27. Image after applied a 13x13 blur kernel on the original image (GPU)

Execution time (ms): 0.325 At the middle right of the panel we can see the execution time is

0.325 milliseconds in GPU. This time is the average time after we run 100 times of this operation. Convolution

Convolution

Definition

A convolution kernel, convolution matrix, or mask is a small matrix used for blurring, sharpening, edge detection, and more. This is accomplished by means of convolution between a kernel and an image.

Demo

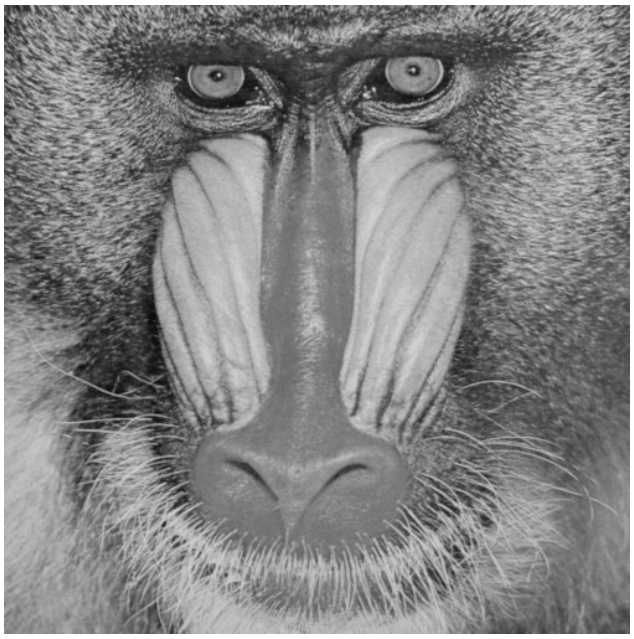


Fig. 28. Original Input Image

```
.02 .02 .02 .02 .02 .02 .02
.02 .02 .02 .02 .02 .02 .02
.02 .02 .02 .02 .02 .02 .02
.02 .02 .02 .02 .02 .02 .02
.02 .02 .02 .02 .02 .02 .02
.02 .02 .02 .02 .02 .02 .02
.02 .02 .02 .02 .02 .02 .02
```

(a)7x7 blur Kernel values

```
-5  0  5
-5  0  5
-5  0  5
```

(b) KernelEdgeX

```
-5  -5  -5
 0   0   0
 5   5   5
```

elLaplacian

(c)KernelEdgeY

```
-1  -1  -1
-1   8  -1
-1  -1  -1
```

(d) KernelLaplacian

Fig. 29. Kernels

In Fig. 10, (a) is the kernel values applying on the target image will blur that image. (b) is the kernel values applying on the target image will show the edges of the target image along the horizontal direction. (c) is the kernel values applying on the target image will show the edges of the target image along the vertical direction. (d) is the kernel values applying on the target image will show the edges of the target image along both horizontal and vertical directions.

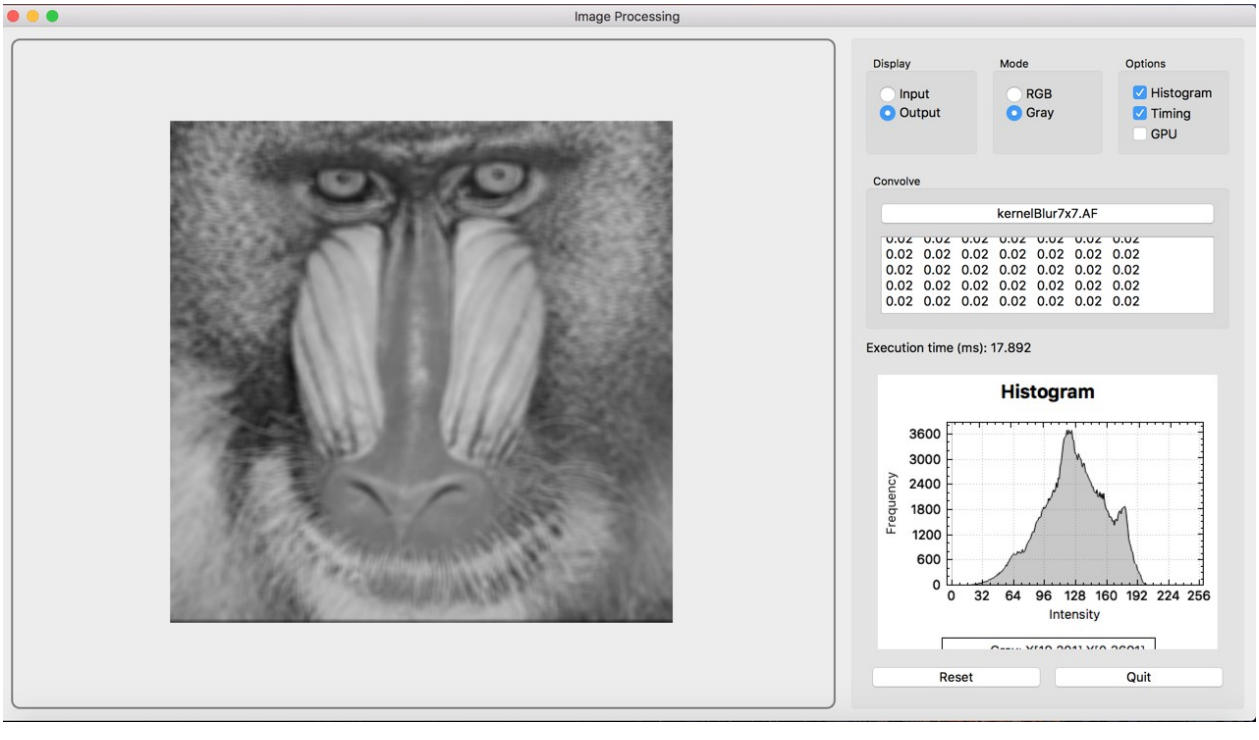


Fig. 30. Image after apply a 7x7 Blur kernel to the input image in CPU

```
Execution time (ms): 17.892
```

At the middle right of the panel we can see the execution time in CPU is 17.892 milliseconds.

This time is the average time after we run 100 times of this operation.

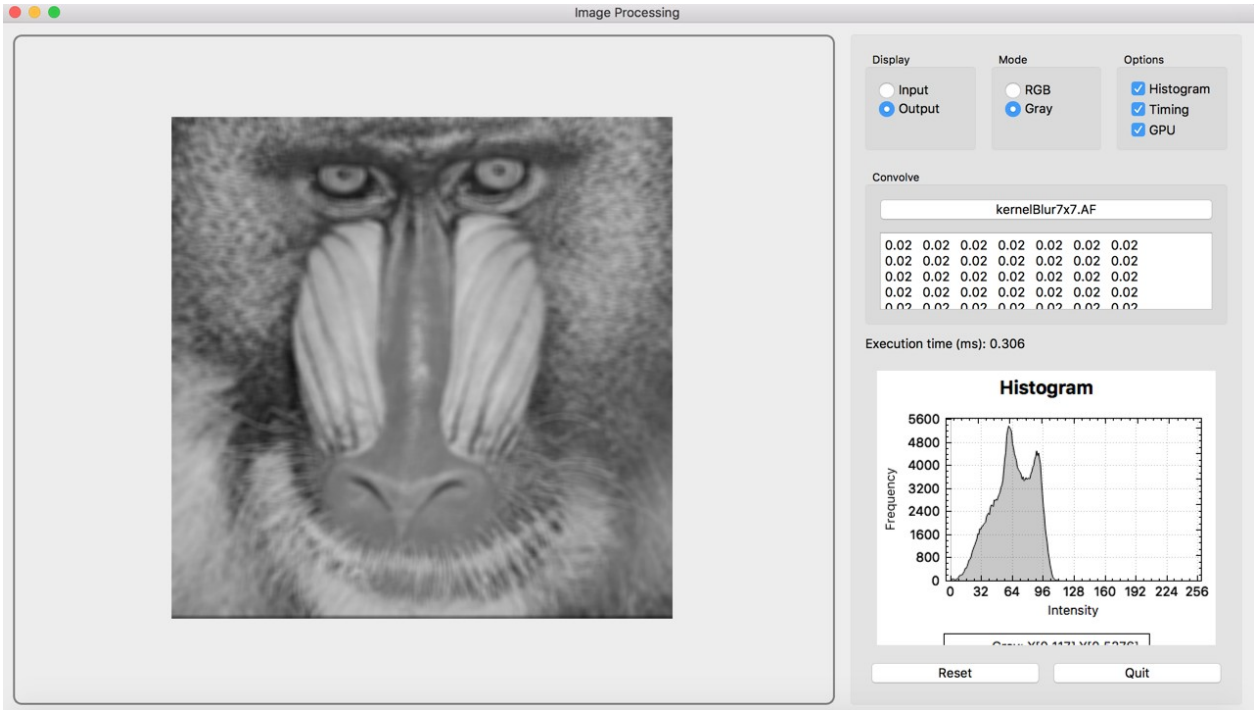


Fig. 31. Image after apply a 7x7 Blur kernel to the input image in GPU

```
Execution time (ms): 0.306
```

At the middle right of the panel we can see the execution time in GPU is 0.306 milliseconds.

It means image rendering in GPU is faster than in CPU.

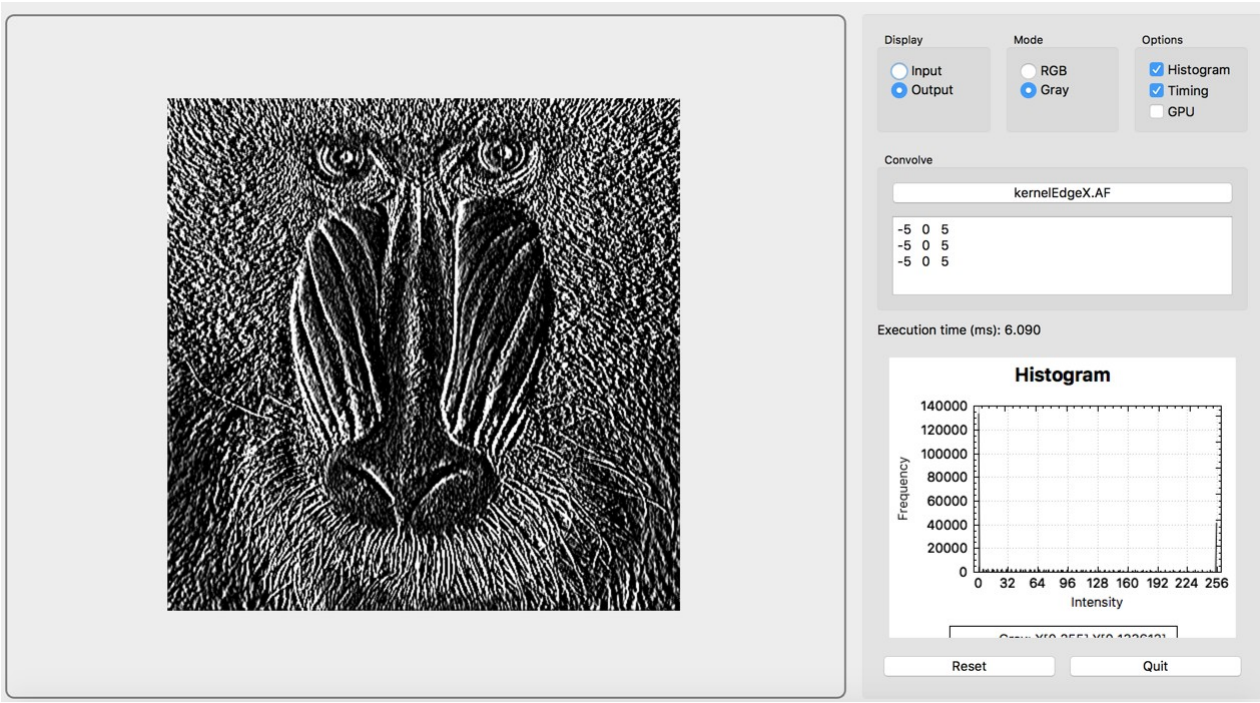


Fig. 32. Image after apply a KernelEdgeX to the input image (CPU)

Execution time (ms): 6.090

At the middle right of the panel we can see the execution time in CPU is 6.090 milliseconds.

This time is the average time after we run 100 times of this operation.

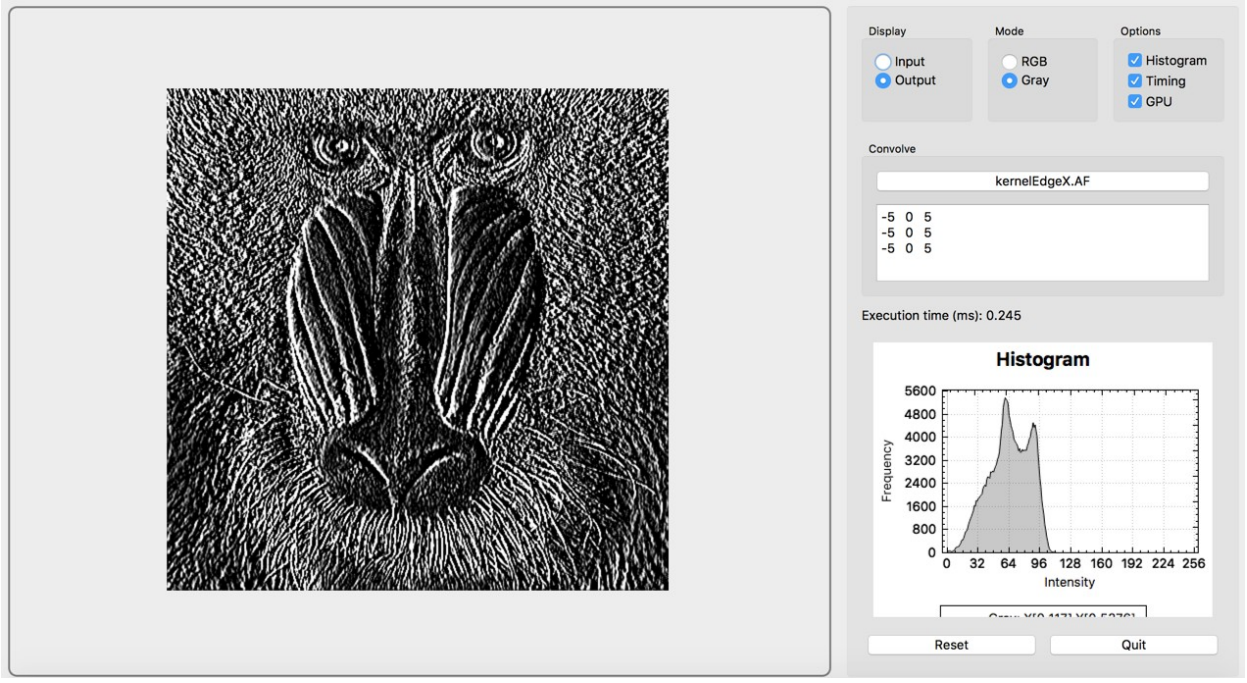


Fig. 33. Image after apply a KernelEdgeX to the input image (GPU)

```
Execution time (ms): 0.245
```

At the middle right of the panel we can see the execution time in GPU is 0.245 milliseconds.

It means image rendering in GPU is faster than in CPU.

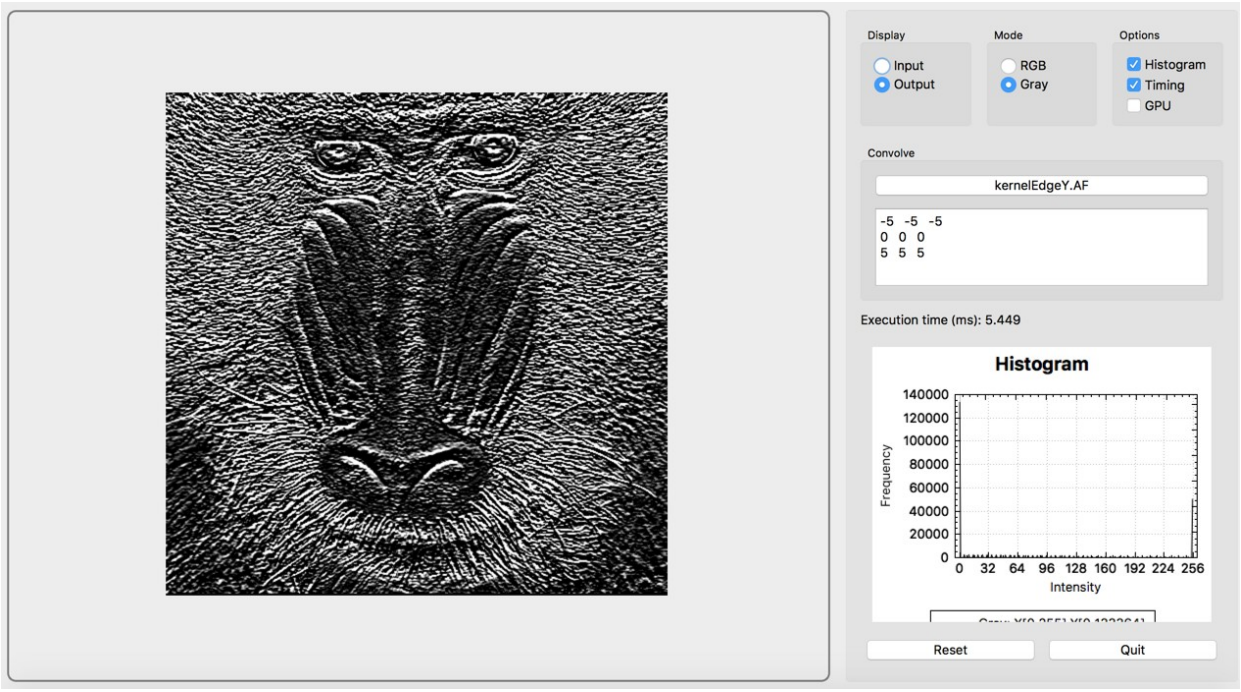


Fig. 34. Image after apply a KernelEdgeY to the input image (CPU)

```
Execution time (ms): 5.449
```

At the middle right of the panel we can see the execution time in CPU is 5.449 milliseconds.

This time is the average time after we run 100 times of this operation.

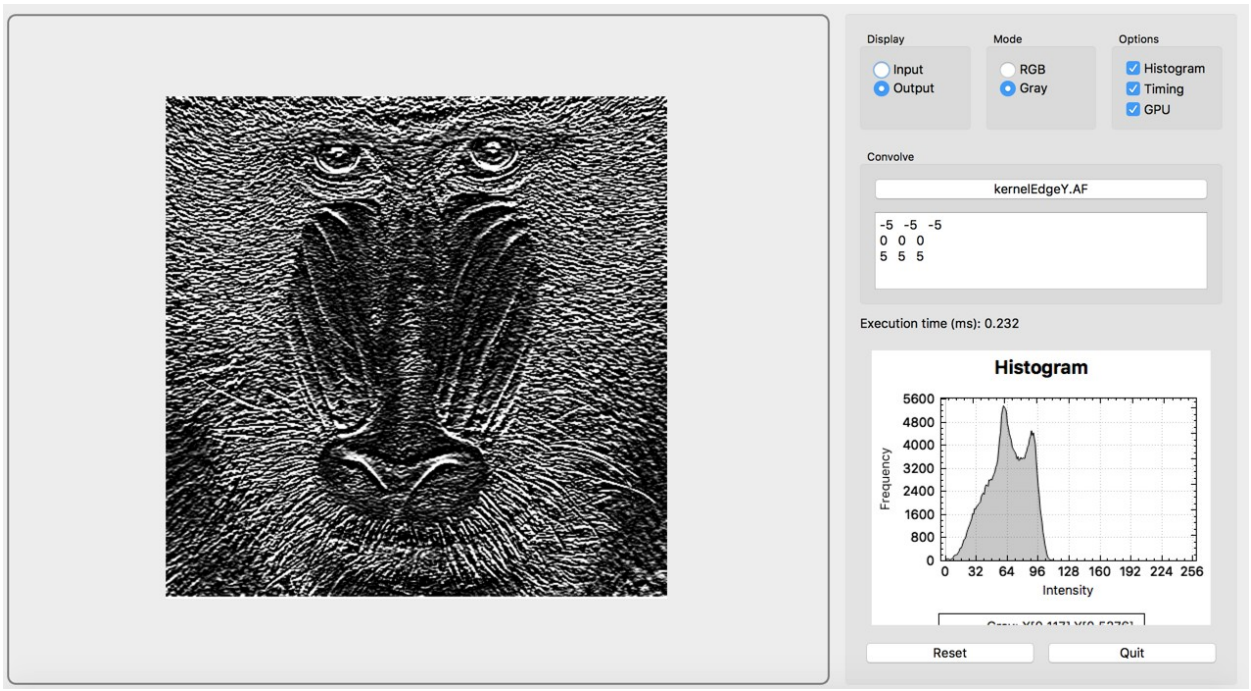


Fig. 35. Image after apply a KernelEdgeY to the input image (GPU)

Execution time (ms): 0.232

At the middle right of the panel we can see the execution time in GPU is 0.232 milliseconds.

It means image rendering in GPU is faster than in CPU.

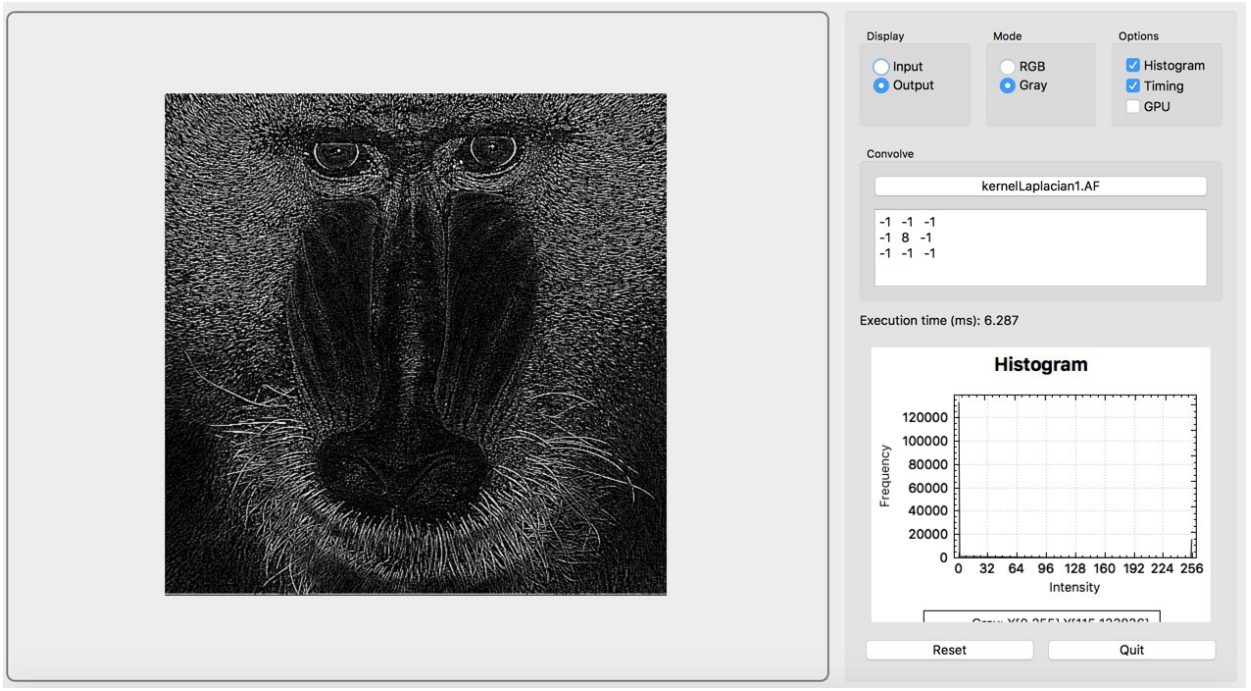


Fig. 36. Image after apply a KernelLaplacian to the input image (CPU)

Execution time (ms): 6.287

At the middle right of the panel we can see the execution time in GPU is 6.287 milliseconds.

This time is the average time after we run 100 times of this operation.

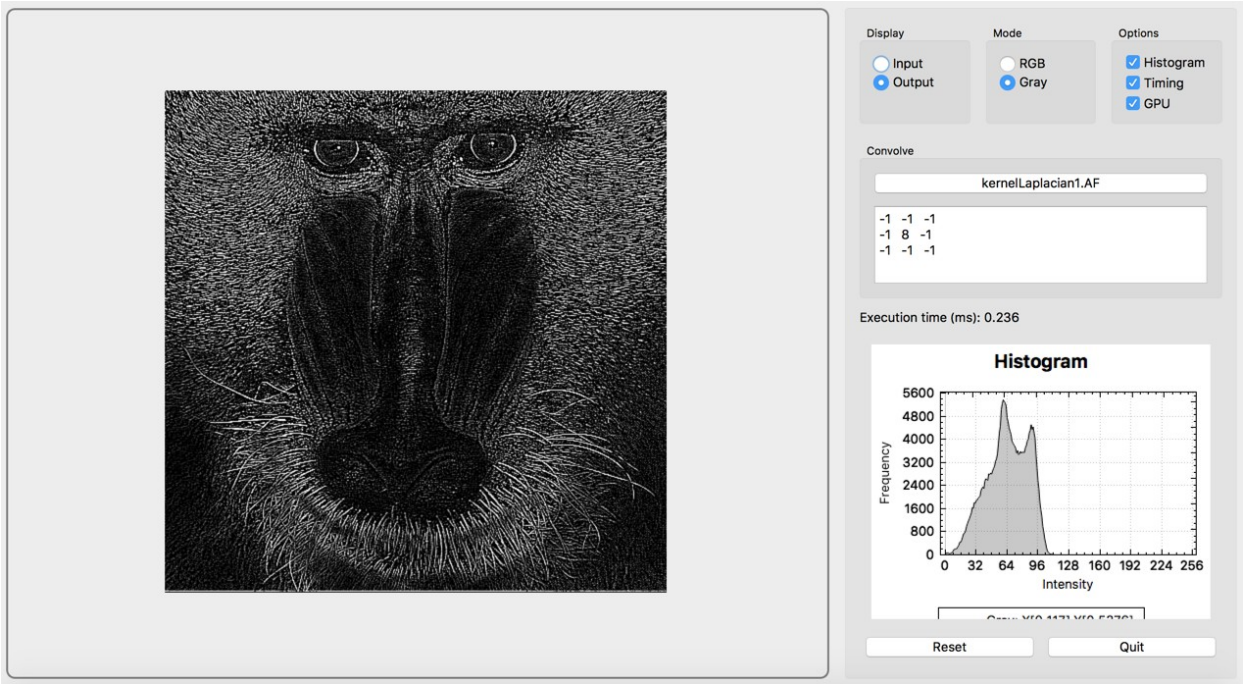


Fig. 37. Image after apply a KernelLaplacian to the input image (GPU)

Execution time (ms): 0.236

At the middle right of the panel we can see the execution time in GPU is 0.236 milliseconds.

It means image rendering in GPU is faster than in CPU.

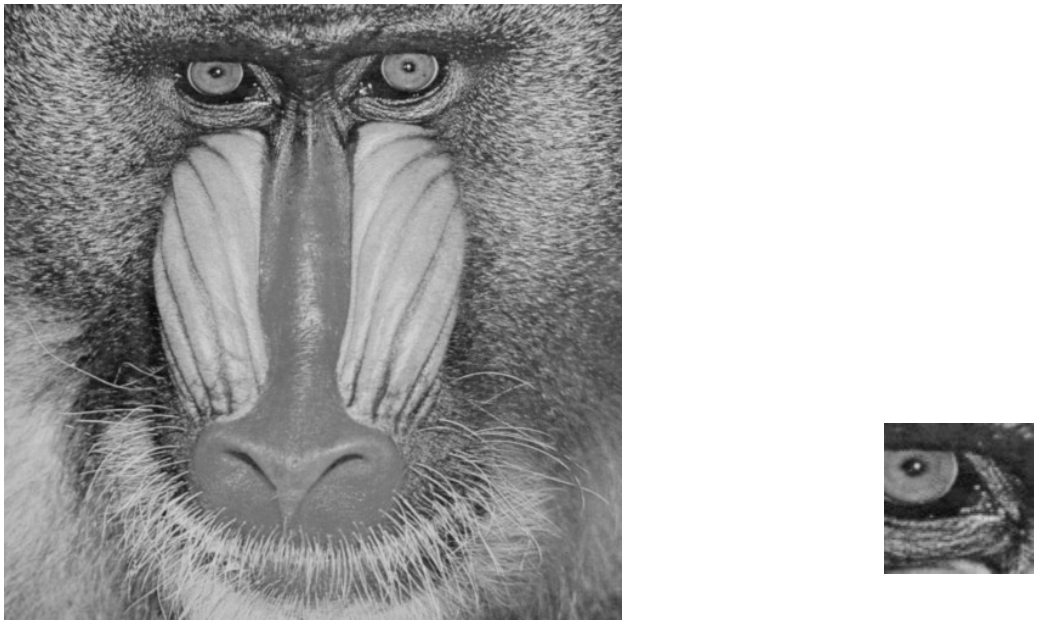
This kernel applying on the target image will show the edges of the target image.

Correlation

Definition

Correlation is a measure of similarity between two images. In our case, given a template image, we will find the most likely matched part in the target image.

Demo



(a)

(b)

Fig. 38

Fig. 19. Two images we will use for the first demo. Left side (a) is the target image and right side

(b) is the template image cropped from the target image, which is the right eye of the mandril.

The correlation has two steps in this implementation. Look at the two steps below, it first shows an intermediate image during our correlation operation and then when we click the match button the template image will find its best match position on the target image

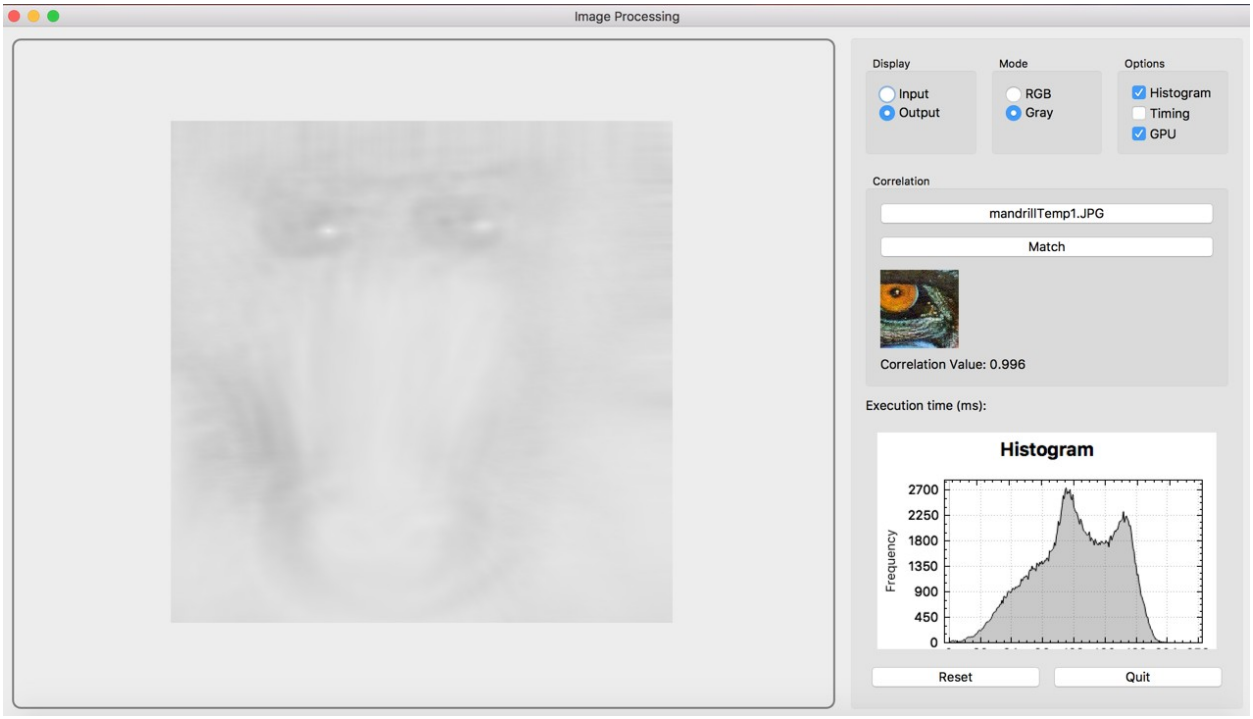


Fig. 39. Image after the correlation values assigned on the output pixels (GPU)

Form Fig. 15 above we can see the brightest point is at the right eye of the mandril, after we click the match button the template image should match there.

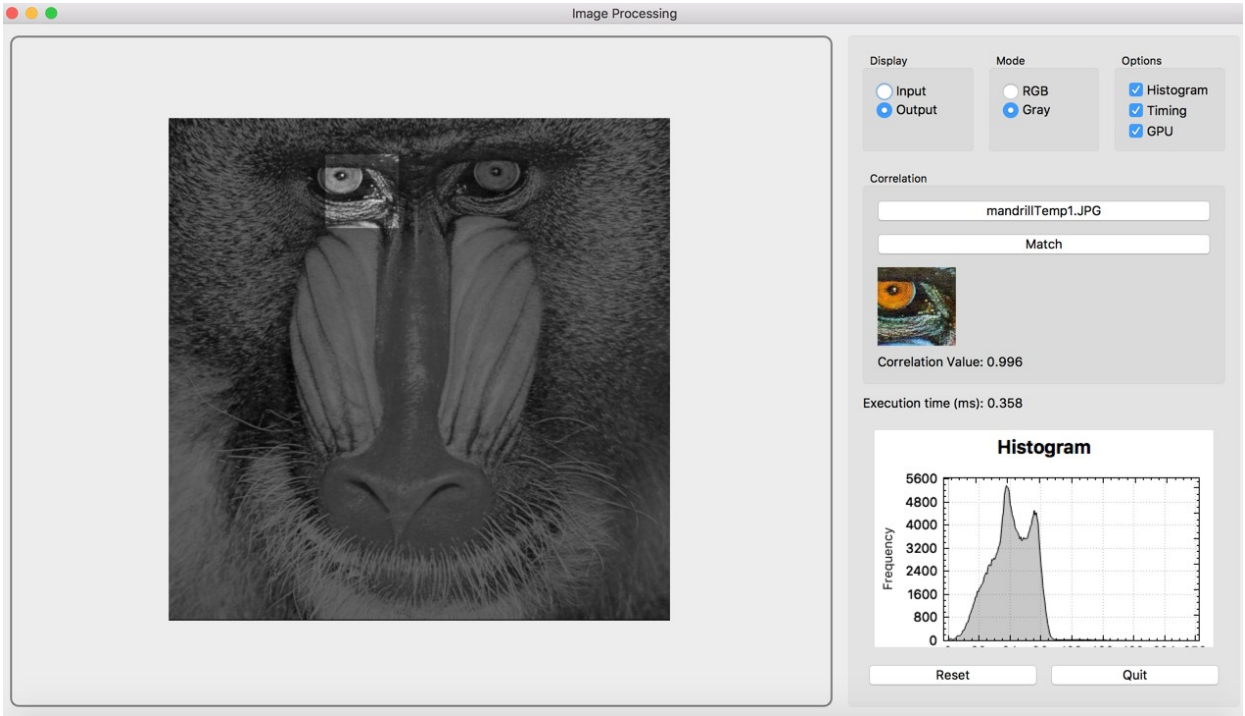
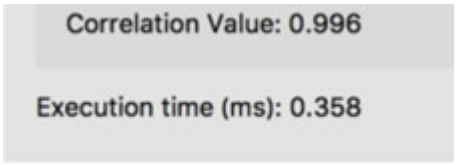


Fig. 40. Image after the template image matched on the target image (GPU)



At the middle right of the panel we can see the maximum correlation value is 0.996 and the time cost of the correlation operation to find the best match is 0.358 milliseconds. This time is the average time after we run 100 times of this operation.

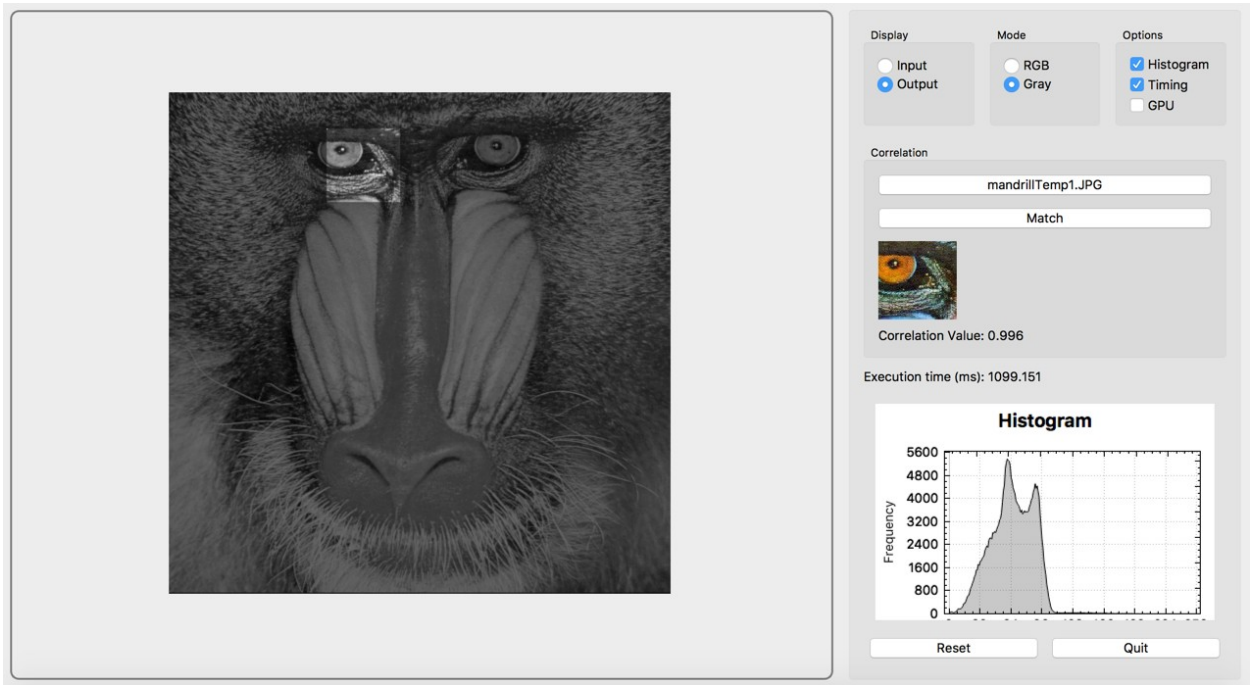
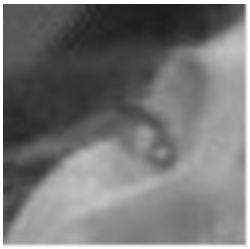


Fig. 41. Image after the template image matched on the target image (CPU)

At the middle right of the panel we can see the execution time in CPU is 1099.151 milliseconds. This time is the average time after we run 100 times of this operation. The performance is so bad compared to the GPU rendering.



(a) (b)

Fig. 42

Fig. 23. Two images we will use for the second demo. Left side image (a) is the target image and right side image (b) is the template image cropped from the target image, which is the right eye of the maid.

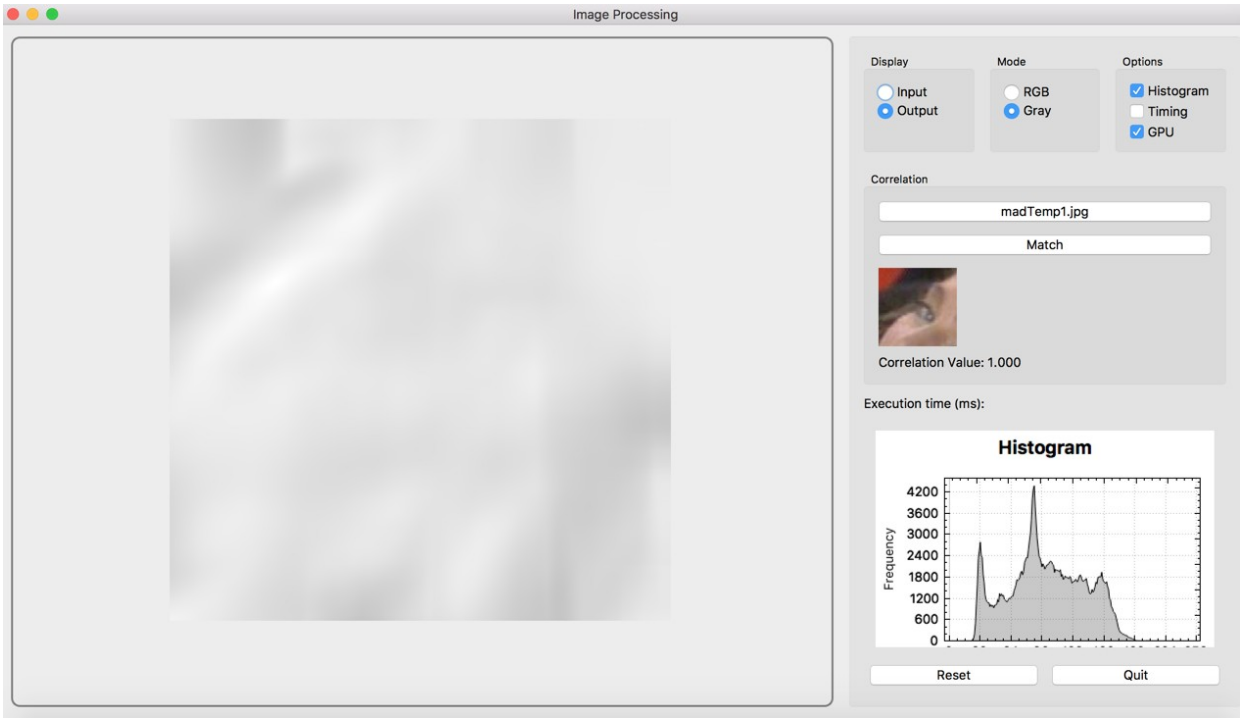


Fig. 43. Image after the correlation values assigned on the output pixels

From Fig. 18 above we can see the brightest point is at the right eye of the maid, after we click the match button the template image should match there.

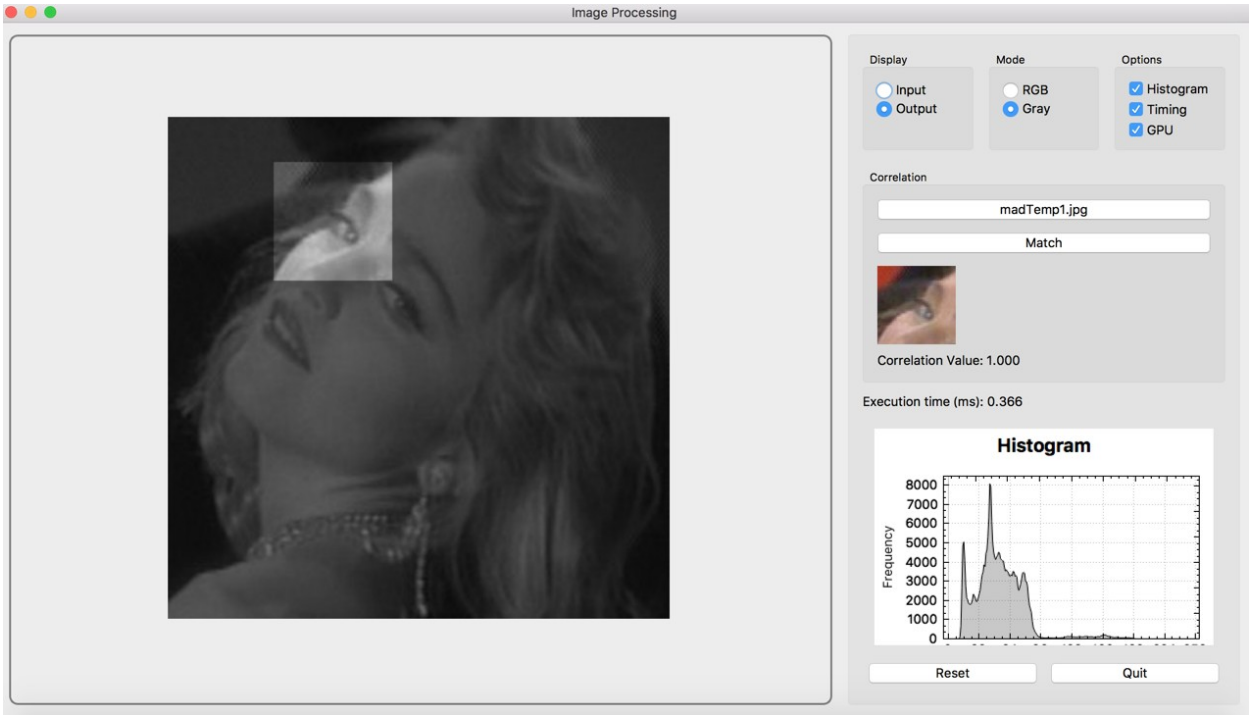


Fig. 44. Image after the template image matched on the target image