Imagine - Programacion sobre redes

Tobias Gutierrez, Tiago Maetokuhiga, Fernando Reynoso y Nicolás Maruyama

29 de Mayo de 2023

Contents

1 Introducción **1**

2 Desarrollo **2**
- 2.1 Black and White . . . 2
  - 2.1.1 Single-Thread . . . 2
  - 2.1.2 Multi-Thread . . . 2
- 2.2 Shades . . . 2
  - 2.2.1 Single-Thread . . . 2
  - 2.2.2 Multi-Thread . . . 3
- 2.3 Brightness . . . 3
  - 2.3.1 Single-Thread . . . 3
  - 2.3.2 Multi-Thread . . . 3
- 2.4 Contrast . . . 3
  - 2.4.1 Single-Thread . . . 3
  - 2.4.2 Multi-Thread . . . 4
- 2.5 Merge . . . 4
  - 2.5.1 Single-Thread . . . 4
  - 2.5.2 Multi-Thread . . . 5
- 2.6 Box Blur . . . 5
  - 2.6.1 Single-Thread . . . 5
  - 2.6.2 Multi-Thread . . . 5
- 2.7 Edge Detection . . . 5
  - 2.7.1 Single-Thread . . . 5
  - 2.7.2 Multi-Thread . . . 6
- 2.8 Sharpen . . . 6
  - 2.8.1 Single-Thread . . . 6
  - 2.8.2 Multi-Thread . . . 7

3 Experimentos **8**

4 Conclusiones **9**

A Redes Neuronales **10**

Bibliografia **10**

Chapter 1

Introducción

En el presente trabajo implementamos filtros a imagenes utilizando el formato *Portable Pixel Map* (.ppm) y el lenguaje *C++*. El trabajo consta de cuatro etapas distintas:

En la primera etapa, nos enfocamos en la implementación de los filtros de modo single-thread. En la segunda etapa, nos centramos en la implementación multi-thread. En la tercera etapa, nos dedicamos a la experimentación e informe. Realizamos una serie de pruebas, en cuanto al desarollo de nuevos filtros, ademas de la comprension del funcionamiento interno de cada filtro. Como utlima etapa presentamos los resultados y conclusiones en el vigente informe. El informe muestra los resultados obtenidos a lo largo de la implementacion, experimentacion, asi como una explicacion sobre los mismos.

Chapter 2

Desarrollo

2.1 Black and White

2.1.1 Single-Thread

El filtro consiste en transformar una imagen a escala de grises. Como primer paso fundamental debimos comprender el concepto fundamental del filtro. En el cual para cada píxel de la imagen, necesitamos mapear su color a un tono de gris. La intensidad de este gris se determina calculando el promedio de los canales rojo, verde y azul del píxel. Una vez comprendido el concepto del fitlro pasamos directo a la implementacion. Basicamente la funcion consiste en iterar la imagen para obtener la cantidad de pixeles tanto en altura como en ancho. Luego para cada pixel en la posicion *(i, j)*, utilizamos el metodo *getpixel*. Este objeto pixel, contiene los valores de cada canal rgb. A continuacion se calculo la intensidad del gris, a partir del promedio de la suma de los valores rgb de cada pixel. Con estos valores calculados, seteamos un nuevo objeto pixel llamado *np*, donde los valores tres valores rgb, son iguales, siendo el promedio anteriormente calculado. Finalmente con el metodo *setpixel*, reemplazamos los pixeles orginiales con el nuevo píxel *np*, que ahora tiene los valores de los canales *RGB* iguales y representan el tono de gris correspondiente.

2.1.2 Multi-Thread

2.2 Shades

2.2.1 Single-Thread

El filtro consiste en convertir la imagen a una escala de grises con n tonalidades diferentes. En primer lugar, analizamos detenidamente la consigna y comprendimos el enfoque general del filtro Shade. A diferencia del filtro *Black and White*, en este caso queremos asignar varias intensidades de color a una misma tonalidad de gris. Al comienzo de la funcion repetimos la iteracion para recorrer todo los pixeles de la imagen

2.2.2 Multi-Thread

2.3 Brightness

2.3.1 Single-Thread

El filtro de brightness funciona aumentando o disminuyendo la intensidad de los colores de una imagen. Para cada píxel de la imagen, representado como $I(x, y) =< r, g, b >$, aplicamos una transformación utilizando el porcentaje de brillo (b) como factor. La fórmula de transformación es la siguiente:

$I(x, y) =< r + 255 * b, g + 255 * b, b + 255 * b >$, donde b $\in [-1, 1]$.

En la primera parte de la implementacion, volvimos a recorrer todos los pixeles de la imagen, tanto en ancho como en altura. Dentro del bucle, obtenemos el píxel actual utilizando la función *getPixel(i, j)* de la imagen. Guardamos los valores de los canales de color en variables separadas: *nr* para el canal rojo, *ng* para el canal verde y *nb* para el canal azul. A continuación, aplicamos la fórmula de transformación del brillo a cada canal de color. Multiplicamos 255 por el valor del brillo (representado por la variable b) y lo sumamos al valor original del canal respectivo. Luego, como anteriormente, creamos un nuevo objeto de píxel llamado *np* con los nuevos valores de los canales de color calculados. Antes de continuar, es importante destacar que los valores resultantes pueden exceder el rango permitido de intensidades ([0, 255]). Por lo tanto, necesitamos asegurarnos de que los valores se mantengan dentro del rango establecido. Para hacer esto, llamamos a la función *truncate()* del objeto de píxel *np*, que truncará cualquier valor fuera del rango permitido. Finalmente, actualizamos el píxel en la posición actual de la imagen de entrada utilizando la función *setPixel(i, j, np)*.

2.3.2 Multi-Thread

2.4 Contrast

2.4.1 Single-Thread

Primero, consideramos la representación de un píxel en la imagen como $I(x, y) =< r, g, b >$, donde r, g y b corresponden a los valores de intensidad de los canales rojo, verde y azul, respectivamente. Luego, aplicamos la siguiente transformación a cada píxel:

$I(x, y) =< \lambda * (r - 128) + 128, \lambda * (g - 128) + 128, \lambda * (b - 128) + 128 >$

Aquí, λ se calcula utilizando el factor de intensidad del contraste c. El factor c no está normalizado como en el caso anterior y puede variar dentro del rango $[-225, 255]$. El numero λ se define como:

$\lambda = \frac{259(c+255)}{255(259-c)}$

Es importante tener en cuenta que, después de aplicar estas operaciones, los valores

resultantes de cada píxel pueden superar el límite superior de 255 o ser menores que 0. Para la implementacion recibimos el valor del factor de intensidad de contraste a través de la variable *contrast*, que se obtiene de los argumentos pasados al programa.

Luego, calculamos el factor de intensidad de contraste c utilizando la fórmula mencionada anteriormente.

A continuación, implementamos dos bucles for anidados para recorrer cada píxel de la imagen. Dentro de los bucles, obtenemos el píxel actual utilizando la función *getPixel(i, j)* y almacenamos los valores de los canales de color en las variables correspondientes: r para el canal rojo, g para el canal verde y b para el canal azul.

Luego, aplicamos la transformación del contraste a cada canal de color utilizando la fórmula mencionada anteriormente.

Después de obtener los nuevos valores de los canales de color, creamos un nuevo objeto de píxel llamado *np* con estos valores.

Para asegurarnos de que los valores de los canales de color se mantengan dentro del rango permitido, llamamos a la función *.truncate()* del objeto de píxel *np*, que trunca cualquier valor que supere el límite superior de 255 o sea menor que 0.

Finalmente, actualizamos el píxel en la posición actual de la imagen de entrada utilizando la función *setPixel(i, j, np)*.

Este enfoque nos permite modificar el contraste de la imagen de manera efectiva y garantizar que los valores de los píxeles se mantengan dentro del rango válido.

2.4.2 Multi-Thread

2.5 Merge

2.5.1 Single-Thread

En el filtro de merge comprendimos que debíamos realizar la mezcla de los canales de color rojo (r), verde (g) y azul (b) de los píxeles individuales. Continuando con la implementación, convertimos el porcentaje de mezcla proporcionado (p1) de tipo string a tipo float utilizando la función atof(). Esto nos permitió trabajar con valores decimales para calcular la ponderación de los colores en la mezcla.

A continuación, comenzamos a iterar sobre los píxeles de la primera imagen utilizando dos bucles anidados. Para cada posición (x, y) en la imagen, obtuvimos los valores de los píxeles correspondientes tanto de la imagen 1 como de la imagen 2. Estos valores se almacenaron en objetos de tipo pixel.

Luego, procedimos a calcular los valores de los canales de color (r, g, b) para el píxel resultante de la mezcla. Utilizamos la fórmula dada en el enunciado, multiplicando los valores de los canales de color de la primera imagen (pixel1) por el porcentaje de mezcla

(p1) y los valores de los canales de color de la segunda imagen (pixel2) por el complemento del porcentaje de mezcla (1 - p1). Estos valores ponderados se sumaron para obtener los nuevos valores de los canales de color.

Finalmente, creamos un nuevo objeto de tipo pixel con los valores calculados y actualizamos el píxel correspondiente en la primera imagen (img1) con este nuevo valor mezclado utilizando la función setPixel().

De esta manera, logramos implementar el código necesario para realizar la mezcla de las dos imágenes. El proceso se repite para cada píxel de la imagen, asegurando que cada uno se mezcle adecuadamente según el porcentaje especificado.

2.5.2 Multi-Thread

2.6 Box Blur

2.6.1 Single-Thread

En primer lugar, nos familiarizamos con el enunciado del problema y comprendimos que debíamos utilizar un kernel específico para aplicar el desenfoque. El kernel consistía en una matriz 3x3 con todos los elementos iguales a 1/9, lo que nos permitiría calcular el promedio de los nueve píxeles vecinos. Con los dos bucles anidados para recorrer todos los píxeles de la imagen, excepto los bordes. Comenzamos desde el píxel de coordenadas (1, 1) y continuamos hasta *(img.height - 1, img.width - 1)*. Para cada píxel en esta región, creamos un objeto de píxel llamado *np* para almacenar el resultado del desenfoque.

Dentro de los bucles, utilizamos otro array llamado *pixels* para almacenar los nueve píxeles vecinos correspondientes a cada píxel en el ciclo actual. Utilizamos la función *getPixel()* para obtener estos píxeles de la imagen original.

A continuación, implementamos un bucle adicional para recorrer los nueve píxeles vecinos y realizar el cálculo del promedio ponderado. Para cada píxel vecino, multiplicamos su valor por el elemento correspondiente del kernel y luego lo agregamos al objeto de píxel *np* utilizando la función *addp()*.

Una vez que calculamos el promedio ponderado de los nueve píxeles vecinos, utilizamos la función setPixel() para actualizar el valor del píxel actual en la imagen con el valor resultante almacenado en *np*.

2.6.2 Multi-Thread

2.7 Edge Detection

2.7.1 Single-Thread

En primer lugar, comprendimos que debíamos utilizar el kernel de Sobel para aplicar el filtro y detectar los bordes. A continuación, definimos dos arrays de enteros llamados

kernel y *kernel_t* que almacenaban los valores de los kernels de Sobel para los bordes verticales y horizontales, respectivamente.

Luego, implementamos dos bucles anidados para recorrer todos los píxeles de la imagen, excepto los bordes. Comenzamos desde el píxel de coordenadas (1, 1) y continuamos hasta (img.height - 1, img.width - 1). Para cada píxel en esta región, creamos objetos de píxel llamados *np*, *temp* y *temp_d* para almacenar los resultados intermedios del cálculo.

Dentro de los bucles, utilizamos otro array llamado *pixels* para almacenar los nueve píxeles vecinos correspondientes a cada píxel en el ciclo actual. Utilizamos la función getPixel() para obtener estos píxeles tanto de la imagen original (*image*) como de la imagen en escala de grises después de aplicar el filtro Black White (*image_s*).

A continuación, implementamos un bucle adicional para recorrer los nueve píxeles vecinos y realizar el cálculo de la convolución con los kernels de Sobel. Para cada píxel vecino, multiplicamos su valor por el elemento correspondiente del kernel y luego lo agregamos a los objetos *temp* y *temp_d* utilizando la función *addp()*.

Una vez que calculamos las convoluciones verticales y horizontales, realizamos la combinación de los resultados según la fórmula mencionada en el enunciado:

$I(x, y) = sqrt(Gx^2 + Gy^2)$

En nuestro código, elevamos al cuadrado los valores de los canales de color de los objetos *temp* y *tempd*, y luego los sumamos para obtener *np*. A continuación, truncamos el valor final del píxel tomando la raíz cuadrada de *np.r* y asignándola a los canales de color *np.g* y *np.b*.

Finalmente, utilizamos la función setPixel() para actualizar el valor del píxel actual en la imagen original (*image*) con el valor resultante almacenado en *np*.

2.7.2 Multi-Thread

2.8 Sharpen

2.8.1 Single-Thread

Primero que nada definimos el kernel utilizando un arreglo de enteros llamado kernel. Los valores del kernel se asignaron en el orden correcto según su posición en la matriz.

Luego, implementamos un bucle anidado para recorrer cada píxel de la imagen, excluyendo los bordes. Para cada píxel, creamos un objeto np de la clase pixel para almacenar el nuevo valor del píxel después de aplicar el filtro.

Dentro del bucle, creamos un arreglo de objetos pixels que representan los 9 píxeles vecinos del píxel actual. Utilizamos los métodos getPixel de la clase ppm para obtener los valores de los píxeles vecinos.

A continuación, implementamos otro bucle para multiplicar cada píxel vecino por el valor correspondiente del kernel y agregarlo al objeto np utilizando el método addp de la clase pixel. Esto nos permitió calcular el nuevo valor del píxel actual.

Después de realizar las operaciones de multiplicación y adición con todos los píxeles vecinos, truncamos el valor final del píxel utilizando el método truncate de la clase pixel. Esto aseguró que el valor estuviera dentro del rango válido.

Finalmente, utilizamos el método setPixel de la clase ppm para asignar el nuevo valor del píxel al objeto img en la posición correspondiente.

De esta manera, logramos aplicar el filtro de sharpen a la imagen, resaltando los detalles y bordes gracias al kernel cuidadosamente seleccionado.

2.8.2 Multi-Thread

hola buenas

Chapter 3

Experimentos

Chapter 4

Conclusiones

Appendix A

Redes Neuronales

Bibliography

[1] Nakamoto, Satoshi (2008, 31 de Octubre) *Bitcoin: A Peer-to-Peer Electronic Cash System*, Nakamoto Institute. `https://nakamotoinstitute.org/bitcoin/`