Sistemas Embebidos
Procesamiento Digital de Imágenes

Garcia Lomeli Abraham Amos

14 de octubre de 2018

Índice

1. Introducción

Dentro del cómputo, en ocasiones se vuelve necesario el llevar a cabo diferentes algoritmo de procesamiento digital de imágenes ya sean obtenidas mediante un sensor de imagen o desde un archivo previamente creado.

Algoritmos sumamente utilizados en la actualidad como pueden ser la detección de minucias en huellas digitales o los bordes basan sus principios en la capacidad de la computadora en leer y entender los formatos de de imagen. Para ello a continuación explicaré un breve código en donde realizo las siguientes acciones:

- Obtengo la información de una imagen en formato *BMP*, es decir un *mapa de bits.*
- Convierto la imagen de color a *blanco y negro*
- Aplico un filtrado de detección de bordes
- Aplico un filtrado de tipo Gaussiano
- Convierto de *blanco y negro* a color
- Guardo la imagen ya procesada

Con la finalidad de optimizar el tiempo de ejecución y aprovechar los recursos de la computadora, se han añadido hilos a los filtrados *gaussiano* y de *detección de bordes*

Para lo anterior hago uso de cinco diferentes archivos de código en lenguaje ANSI C:

- ***imagen.h:*** Contiene los datos *struct* en donde se almacena la información de las cabezeras del formato *.bmp*, además se especifican los prototipos de las funciones de filtrado.
- ***imagen.c:*** Contiene la implementación del los filtros
- ***nodito.h:*** Contiene la especificación de la estructura que envio a cada hilo
- ***convercion.c*** Aqui se encoentra la función main del programa
- ***Makefile*** Usado para agilizar la compilación del proyecto.

2. Estructuras en *imagen.h*

El código en *imagen.h* incluye las siguientes estructuras:

```
1typedef struct bmpFileHeader
2{
3
4  uint32_t size;
5  uint16_t resv1;
6  uint16_t resv2;
7  uint32_t offset;
8} bmpFileHeader;
9
10typedef struct bmpInfoHeader
11{
12  uint32_t headersize;
13  uint32_t width;
14  uint32_t height;
15  uint16_t planes;
16  uint16_t bpp;
17  uint32_t compress;
18  uint32_t imgsize;
19  uint32_t bpmx;
20  uint32_t bpmy;
21  uint32_t colors;
22  uint32_t imxtcolors;
23} bmpInfoHeader;
```

En donde *width* y *height* son altura y anchura de la imagen, *imgsize* es el tamaño de la imagen y *bpp* son los bits por pixel.

3. Abrir la imagen

Para poder abrir la imagen se hace uso de un apuntador a *unsigned char* en donde cada pixel será un caracter. En este proyecto se hace uso de la función *abrir BMP* la cual obtiene como parámetros:

- Nombre del archivo en formato *bmp*
- La estructura de tipo *bmpInfoHeader* donde guardará la información de la foto

La función regresa un apuntador de tipo *unsigned char* en donde se guarda la foto sin cabezera:

```
imgdata = (unsigned char *)malloc( bInfoHeader->imgsize );
```

Para visualizar la información de la cabecera del archivo uso la función *displayInfo*.

4. Convertir a escala de grises

Todo lo que se describe en esta sección sucede en la función *RGBtoGray*.
En primera instancia se reserva espacio de memoria para la *nueva imagen en escala de grises*:

```
1 imagenGray = (unsigned char *)malloc( width*height*sizeof(unsigned char) );
```

Para convertir la imagen a grises obtendremos el valor de cada pixel R, G y B, para ello usamos dos indices con los que recorremos la imagen:

- ***IndiceGray:*** recorre pixel por pixel
- ***IndiceRGB:*** es *indiceGray* pero recorriendose de 3 en 3 (ya que son tres bits en un RGB). Es decir que *indiceRGB=indicegray*3*

En cada pixel de la nueva imagen en escala de grises ingresaremos el valor de la variable *graylevel* la cual esta dada por:

```
1 grayLevel = (imagenRGB[indiceRGB]*30+
2                 imagenRGB[indiceRGB+1]*59+
3                 imagenRGB[indiceRGB+2]*11)/100;
```

Al final para movernos sobre toda la foto usamos un ciclo *for* anidado:

```
1  for ( y = 0; y < height; y++)
2  {
3    for ( x = 0; x < width; x++)
4    {
5      indiceGray = (width*y)+x; //wiith*y es la fila, x es la columna en memoria lineal
6      indiceRGB =  (indiceGray<<1) + indiceGray; //es el indice normal pero por 3 bytes
7      //arriba hice indiceGray*3 pero con menos ciclos de relojo, ya que hice
           indiceGRay*(2+1)
8      //indiceRGB me ubica en el pixel R del RGB,
9      grayLevel = (imagenRGB[indiceRGB]*30+
10                imagenRGB[indiceRGB+1]*59+
11                imagenRGB[indiceRGB+2]*11)/100;
12
13     //ahora coloco el valor en el pixel
14     imagenGray[indiceGray] = grayLevel;
15   }
16 }
```

5. Filtrado de detección de bordes

Este algoritmo sucede en la función *filtroImagen*.
La función recibe una estructura en donde especifico:

- Apuntador a los datos de la imagen a filtrar
- Apuntador en donde guardar los datos despues del filtro (con espacio de memoria ya reservado:

```
imagenFiltrada = reservarMemoria(info.width, info.height);
```

- Alto de la imagen
- Ancho de la imagen

Para recibir los datos hago un cast para que cada hilo reciba un dato de tipo *nodito*:

```
struct nodito nod= *(struct nodito *)arg;
```

Para los filtros de detección de bordes se requieren dos máscaras que convolucionaran sobre la imagen. Estos filtros son:

```
char GradC[] =
    {-1, -2, -1,
     0, 0, 0,
     1, 2, 1};
 char GradF[] =
    {1, 0, -1,
     2, 0, -2,
          1, 0, -1};
```

Moveremos ambas máscaras por la imagen, por lo que hay que tener cuidado de no salir de los bordes, siendo así se usará un recorrido en x y en y donde el inicio es *cero* y el final será *width-DIMASK* y *height-DIMASK*.
Es importante destacar:

- DIMASK es igual a 3, por lo tanto 3 números antes del final de la imagen hay que parar.
- Para hacer el procesamiento en paralelo, cada hilo tiene un inicio y final de rango, por lo que el ciclo *for* de y usa dichos rangos.

Para cada paso en la convolución de la imagen con las máscaras se hace:

```
for( ym = 0; ym < DIMASK; ym++ )
      {
        for( xm = 0; xm < DIMASK; xm++ )
        {
          indice = ((y+ym)*width + (x+xm));
          conv1 += imagenGray[indice]*GradF[indicem];
          conv2 += imagenGray[indice]*GradC[indicem++];

        }
      }
      conv1 = conv1 / 4;
      conv2 = conv2 / 4;
      conv1 = sqrt(pow(conv1,2)-pow(conv2,2));
      indice = ((y+1)*width + (x+1));
      imagenFiltro[indice] = conv1;
```

Los valores de cada convolución se guardan, se multiplican por 1/4 y se obtiene la distancia euclideana entre ambos. Al final el valor de dicha distancia será el valor del pixel.

6. Filtrado gaussiano

El proceso es análogo al de la sección anterior, no obtante la máscara a usar ahora es:

```
char Gauss[] =
    {1, 2, 1,
     2, 4, 2,
     1, 2, 1};
```

Para la convolución, como ahora solo se tiene una máscara, se hace:

```
conv1 = 0;
      indicem = 0;
      for( ym = 0; ym < DIMASK; ym++  )
      {
        for( xm = 0; xm < DIMASK; xm++ )
        {
          indice = ((y+ym)*width + (x+xm));
          conv1 += imagenGray[indice]*Gauss[indicem++];

        }
      }
      conv1 = conv1 / 16;
      indice = ((y+1)*width + (x+1));
      imagenFiltro[indice] = conv1;
```

En lugar de aplicar la distancia euclideana, solo se multiplica el resultado de la convolución por 1/16 y se aplica el resultado al pixel.

7. Convertir de *escala de grises* a color

En este caso lo que se hace es recorrer la imagen pixel por pixel, al momento de llegar a el pixel i, se añade su valor al pixel en R, en G y en B:

```
for ( y = 0; y < height; y++)
 {
   for ( x = 0; x < width; x++)
   {
     indiceGray = width*y+x; //wiith*y es la fila, x es la columna en memoria lineal
     indiceRGB =  indiceGray * 3; //es el indice normal pero por 3 bytes


     imagenRGB[indiceRGB+0] = imagenGray[indiceGray];
     imagenRGB[indiceRGB+1] = imagenGray[indiceGray];
     imagenRGB[indiceRGB+2] = imagenGray[indiceGray];

     }
 }
```