Administración y Organización de Computadores

Curso 2023-2024

<u>Práctica</u>: *Detección de bordes y contornos en imágenes*

El principal objetivo de esta práctica es completar una aplicación que incluye varias opciones para detectar bordes y contornos en imágenes. La parte de la aplicación que se encuentra ya implementada está escrita en *C++* y se encarga del control principal de las operaciones y de la gestión de la interfaz de usuario. No obstante, este código se apoya en una serie de funciones que se encuentran vacías y cuya implementación es objeto de esta práctica. Dicha implementación debe realizarse en lenguaje ensamblador. La integración de los dos lenguajes se llevará a cabo mediante las facilidades de ensamblado en línea proporcionadas por el compilador *gcc*. La programación en ensamblador se realizará considerando la arquitectura de un procesador Intel o compatible de 64 bits.

A continuación, se detallan diferentes aspectos a tener en cuenta para el desarrollo de esta práctica.

<u>Descripción de la aplicación</u>

El código que deberá ser completado estará incluido en una aplicación *C++*, disponible como proyecto de *Qt*. La siguiente figura muestra una captura de pantalla en un instante de ejecución de la aplicación:

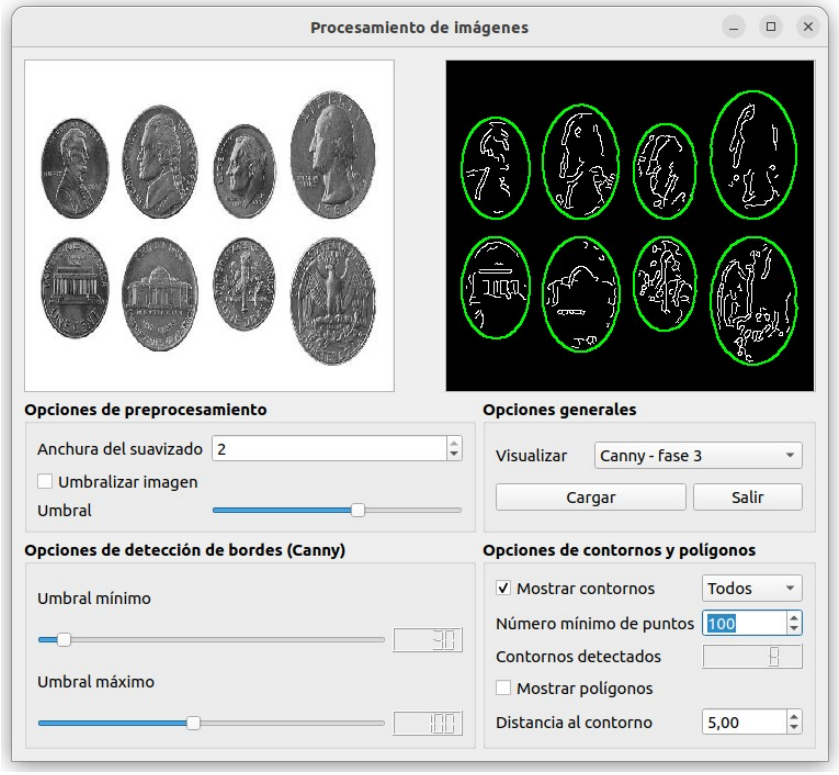


Figura 1: interfaz de la aplicación

¿Qué es Qt?

Qt es un *framework* de desarrollo de aplicaciones que, entre otras aportaciones, proporciona herramientas y librerías de clases para la creación de interfaces de usuario en entornos de escritorio.

Puesta en marcha del proyecto

El fichero que contiene la descripción del proyecto (*pracaoc.pro*) se encuentra disponible en la carpeta principal de la aplicación. Dicho fichero puede ser utilizado para importar el proyecto desde diferentes entornos de desarrollo. No obstante, se recomienda trabajar con Qt Creator (paquete **qtcreator**). Además del paquete **qtcreator**, es necesario instalar los paquetes que contienen las librerías y herramientas de Qt: **qttools5-dev-tools**, **qtbase5-dev**, **qtchooser, qt5-qmake** y **qtbase5-dev-tools**.

Funcionamiento de la aplicación

La aplicación que deberá completarse realiza una serie de operaciones de procesamiento sobre una imagen con el objetivo de obtener los bordes y contornos de los distintos objetos presentes en ella. Las imágenes procesadas por el programa tienen un tamaño variable (puede modificarse redimensionando la interfaz gráfica) y cada píxel está representado por un nivel de gris (0-negro, 255-blanco). En memoria, una imagen se representa como un array bidimensional con un total de *H*W* elementos (siendo *H* el número de filas de la imagen y *W* el número de columnas), en el que cada elemento es de tipo *unsigned char* (1 byte sin signo) para representar los posibles valores en el rango entre 0 y 255.

Las operaciones que se llevan acabo se organizan en 3 bloques:

- **Preprocesamiento**: previo a la detección de bordes y contornos, el usuario puede aplicar 2 operaciones de preprocesamiento sobre la imagen para mejorar los resultados de detección. Estas dos operaciones son:
  - Suavizado: aplica, a cada píxel de la imagen, un suavizado gaussiano. Dicho suavizado consiste en una operación, denominada convolución, entre el entono del píxel y una matriz (*kernel*) que permite realizar una media ponderada del entorno del píxel dando más peso a la zona central que a la zona más alejada del píxel. Concretamente, suponiendo que la imagen de entrada es *imagenO*, la imagen destino *imagenD* y que el tamaño del kernel es (2N+1) x (2N+1), el valor que toma un píxel situado en una fila *y* y columna *x*, se obtiene de la siguiente forma:

$$imagenD[y][x] = \frac{\sum_{j=-N}^{N}\sum_{i=-N}^{N} imagenO[y+j][x+i] * kernel[j+N][i+N]}{\sum_{j=-N}^{N}\sum_{i=-N}^{N} kernel[j+N][i+N]}$$

    El valor de *N* puede seleccionarse en la interfaz gráfica mediante la opción "*Anchura del suavizado*". Los posibles valores están comprendidos entre 0 y 2.

  - Umbralización: genera una imagen en blanco y negro a partir de un valor umbral. Concretamente, los pixels de la imagen origen que tengan un valor menor o igual que el umbral son puestos a 0 (negro) en la imagen destino. Por el contrario, los píxels con un valor superior al umbral en la imagen origen toman un valor de 255 (blanco) en la imagen destino. Para realizar esta operación, el usuario debe activar la opción "*Umbralizar*" de la interfaz y seleccionar el umbral concreto a través de la barra de desplazamiento situada debajo de dicha opción.

- **Detección de bordes**: se dice que un punto de imagen es un punto de borde si existe una diferencia elevada entre el nivel de gris del píxel y el nivel de gris de sus vecinos en alguna dirección. La detección de bordes genera una imagen binaria en la que los puntos de borde toman un valor de 255 (blanco) y los restantes un valor de 0 (negro). Para realizar la detección, se llevará acabo el método de Canny, que divide el proceso en 3 fases:

  - Cálculo de gradientes (fase 1): consiste en calcular, por cada píxel, las diferencias (gradientes) entre vecinos horizontales y verticales en el entorno local del píxel. Esto se resuelve aplicando dos kernels, *kernel_x* y *kernel_y*, (conocidos como operador Sobel) al entorno de 3x3 de cada píxel, siguiendo una operación similar a la del suavizado. Suponiendo que los gradientes se almacenan en una imagen en la que por cada píxel hay dos atributos (*dx* y *dy*), para un determinado píxel situado en una fila *y* y columna *x*, el cálculo de gradientes se resuelve de la siguiente manera:

$$gradientes[y][x].\mathbf{dx} = \sum_{j=-1}^{1}\sum_{i=-1}^{1} imagenO[y+j][x+i] * kernel\_x[j+1][i+1]$$

$$gradientes[y][x].\mathbf{dy} = \sum_{j=-1}^{1}\sum_{i=-1}^{1} imagenO[y+j][x+i] * kernel\_y[j+1][i+1]$$

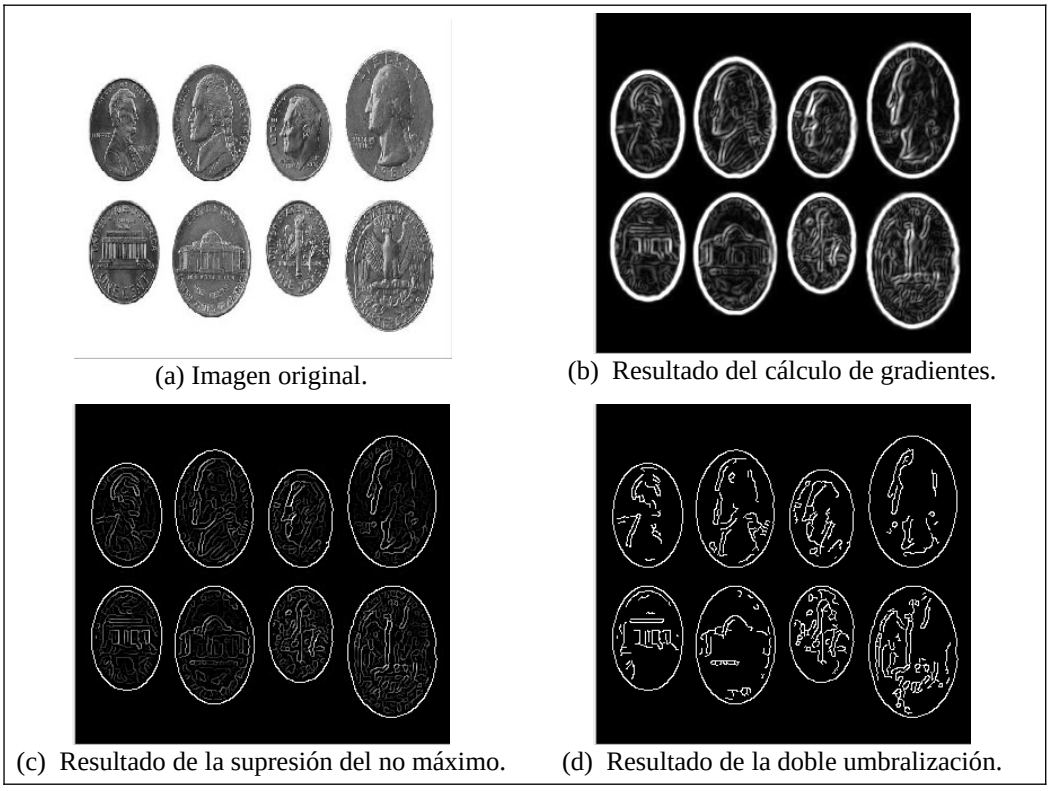
    El resultado visual de esta fase es una imagen (imagen de intensidad de borde) en la que el nivel de gris de cada píxel se calcula como la norma del gradiente horizontal y vertical (sqtr($dx^2 + dy^2$)). Los puntos de borde tendrán un valor alto en esta imagen (nivel de gris claro), mientras que los puntos que no pertenezcan a un borde tendrán valores cercanos al 0 (nivel de gris oscuro). La figura 2(b) muestra un ejemplo de este resultado.

  - Supresión del no máximo (fase 2): a partir de los valores de gradiente horizontal (*dx*) y vertical (*dy*) de cada punto, se puede calcular la norma del gradiente (intensidad del borde, resultado visual de la fase anterior) y su dirección en cada punto (atan(dy/dx)). Utilizando esta idea, en esta fase, se utilizan dos imágenes como entrada: una contiene la norma del gradiente de cada píxel y la otra la dirección del gradiente (dirección del borde: horizontal, vertical, diagonal). Los valores de cada píxel en la imagen de dirección se encuentran en el rango entre 0 y 3 para representar las 4 direcciones posibles: 0 → horizontal; 1 → diagonal izquierda; 2 → diagonal derecha; 3 → vertical. Utilizando estas dos imágenes, en esta fase se obtiene una nueva imagen de intensidad de borde en la que los bordes detectados tienen una anchura de 1 píxel. Para ello, se comprueba si el valor de intensidad de borde (norma del gradiente) de un píxel es superior a la intensidad de borde de los 2 vecinos indicados por la dirección del borde (vecinos horizontales, verticales o diagonales). En tal caso, en la imagen destino, el píxel mantiene su valor de intensidad de borde. Sin embargo, si la intensidad de borde de uno de los dos vecinos es superior a la del píxel actual, dicho píxel es puesto a 0 en la imagen destino. En la figura 2(c) se muestra el resultado de aplicar esta fase sobre una imagen de ejemplo.

  - Doble umbralización (fase 3): en esta última fase, se obtiene una imagen binaria de bordes en la que cada píxel perteneciente a un borde toma un valor de 255 y los restantes píxels un valor de 0. Para ello, se parte de la imagen obtenida en la fase anterior y de dos umbrales (mínimo umbral y máximo umbral). Por cada píxel, se comprueba en primer lugar si su intensidad es superior al valor indicado por el máximo umbral. En ese caso, el píxel se considera punto de borde y se le asigna un valor de 255 en la imagen destino. Si su valor es inferior al del umbral máximo, se comprueba si es superior al umbral mínimo. De ser así, se verifica si el píxel tiene algún vecino con intensidad superior al umbral máximo o que ya haya sido marcado

como punto de borde. Si se cumple alguna de estas dos condiciones, el píxel se marca como punto de borde poniendo un valor de 255 en su posición de la imagen destino. En cualquier otro caso, se considera que el punto no es de borde y se asigna un valor de 0 en esa posición de la imagen destino. La figura 2(d) muestra el resultado de aplicar esta última fase sobre la imagen obtenida del paso anterior.

La selección de los umbrales mínimo y máximo puede llevarse a cabo a través de las opciones disponibles para detección de bordes de la interfaz gráfica.



(a) Imagen original. (b) Resultado del cálculo de gradientes.

(c) Resultado de la supresión del no máximo. (d) Resultado de la doble umbralización.

Figura 2: resultado de cada fase del método de detección de bordes de Canny sobre la imagen indicada en (a).

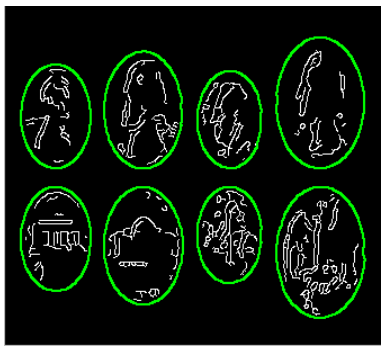
- **Detección de contornos y polígonos**: una vez que se obtiene la imagen de bordes, pueden detectarse los contornos asociados con los objetos que aparecen en ella. Cada contorno consiste en una lista de puntos de borde conectados entre ellos. Tras esta detección, es posible aproximar cada contorno a un polígono para reducir el número de puntos usados para representarlo. A continuación, se describen con más detalles los procesos de detección de contornos y aproximación a polígonos.

    - Obtención de contornos: para obtener los contornos de la imagen se utiliza la imagen de bordes y una imagen en la que se van marcando los puntos que se asignan a un contorno. Esta imagen, a la que llamaremos imagen de "visitados", permite determinar cuando un contorno se ha recorrido completamente. La detección de un contorno parte de la posición de un punto de borde inicial. Ese punto se almacena en la lista de puntos que definirán el contorno. Tras el almacenamiento, se comprueba si el punto está conectado con algún otro punto de borde que no haya sido visitado aún. Esto se comprueba accediendo a las posiciones de los 8 vecinos en la imagen de bordes y la imagen de visitados. Si se verifica que algún punto vecino es punto de borde y aún no ha sido visitado, se toma dicho punto como nuevo punto del contorno y se repite el proceso a partir de él. Si no se puede añadir ningún punto nuevo al contorno, el proceso finaliza.

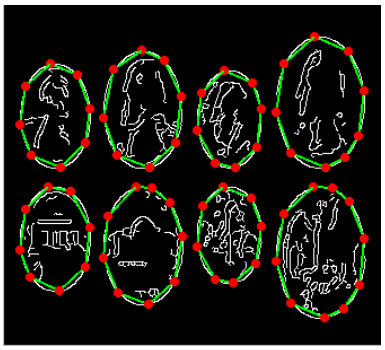
La interfaz gráfica incluye varios elementos relacionados con la detección de contornos dentro del grupo de elementos etiquetados como "*Opciones de contornos y polígonos*". En primer lugar, para visualizar los contornos es necesario activar la opción "*Mostrar contornos"* y seleccionar en el desplegable, situado a la derecha de esta opción, el tipo de contornos que se desea visualizar (abiertos, cerrados o todos). El usuario puede además indicar el mínimo número de puntos que deben tener los contornos detectados mediante otro elemento de la interfaz. Cada contorno se muestra como una cadena de puntos verdes en la imagen de la derecha (ver figura 3(a)). Asimismo, la interfaz incluye otro elemento para indicar el número total de contornos detectados.

- Aproximación de contornos a polígonos: cada contorno puede aproximarse a un polígono siguiendo un procedimiento recursivo. Este procedimiento parte de los puntos inicial y final del contorno. Utilizando estos dos puntos, se define una línea recta que pasa por ellos y se calcula la máxima distancia de los puntos del contorno a dicha recta. Si la máxima distancia detectada es menor que una distancia umbral, se considera que el polígono está definido por esos 2 puntos. Si no, el punto del contorno situado a la máxima distancia de la recta se utiliza para dividir el contorno en 2 (del punto inicial al punto de máxima distancia y del punto de máxima distancia al punto final) y el proceso se repite de manera recursiva para cada uno de esos 2 contornos. Los puntos utilizados para dividir el contorno durante este proceso, junto con los puntos inicial y final, forman los vértices del polígono de aproximación al contorno.

  Dentro del grupo de elementos "*Opciones de contornos y polígonos*", la interfaz incluye una opción para visualizar los polígonos. Si la opción está activa, los polígonos se muestran en la imagen de la derecha de color verde y sus vértices en rojo (ver figura 3(b)). El valor de distancia umbral utilizado durante el proceso de aproximación puede ser modificado a través de otra elemento de la interfaz.



(a) Resultado de la detección de contornos.



(b) Aproximación a polígonos de los contornos.

Figura 3: resultados de la detección de contornos y polígonos de la imagen 2(a).

Además de las opciones ya comentadas, la interfaz gráfica incluye un grupo de opciones generales para cargar una imagen de fichero, visualizar una imagen determinada, dentro del grupo de imágenes obtenidas en cada fase del procesamiento (imagen en negro, imagen suavizada, imagen umbralizada, Canny – fase 1, Canny – fase 2, Canny – fase 3), o salir del programa.

Componentes de la aplicación

El código fuente de la aplicación está formado por 5 ficheros: *main.cpp*, *pracoc.cpp*, *pracoc.h*, *imageprocess.cpp* e *imageprocess.h*. Además, se incluye un formulario de Qt (*mainForm.ui*) y el fichero de descripción del proyecto (*pracaoc.pro*). Junto con estos ficheros, se han incluido

dos imágenes para pruebas: *monedas.jpg* y *poligonos.jpg*. El contenido de cada fichero fuente es el siguiente:

- ***main.cpp***: contiene el procedimiento principal que permite lanzar la aplicación, así como crear y mostrar la ventana principal que actúa como interfaz entre el usuario y la aplicación.
- ***pracaoc.h***: fichero que contiene la definición de la clase principal de la aplicación (*pracAOC*). Esta clase contiene los elementos principales de gestión de la aplicación. Entre los atributos, se encuentran la interfaz de usuario incluida en el programa y las imagenes y listas de puntos que se utilizan para almacenar los contornos y polígonos detectados.
- ***pracaoc.cpp***: Incluye la implementación de los métodos de la clase *pracAOC*. En su mayoría, estos métodos se encargan de responder a los distintos eventos de la interfaz de usuario incluida en el programa, de realizar el procesamiento sobre la imagen de entrada mediante llamadas a las funciones definidas en imageprocess.*h*, y de mostrar gráficamente los resultados.
- ***imageprocess.h***: contiene la definición de las funciones implementadas en el fichero imageprocess.*cpp*. Además, incluye la definición de dos estructuras, denominadas *Gradient* y *Point*. La primera estructura se utiliza para definir los elementos de la imagen de gradientes. Contiene dos atributos, *dx* y *dy*, para almacenar el gradiente horizontal y vertical de cada punto. La segunda estructura, *Point*, incluye dos atributos (*x* e *y*) que representan las coordenadas de un punto en imagen. Esta estructura se utiliza para definir los arrays de puntos que representan los contornos y los polígonos de una imagen.
- ***imageprocess.cpp***: implementación de las funciones de procesamiento de imágenes utilizadas en la clase principal. Todas estas funciones, exceptuando la función *suavizado*, contienen una implementación vacía. El objetivo de esta práctica es completarlas para que el funcionamiento de la aplicación sea el descrito anteriormente.

<u>Extensiones principales en x86-64</u>

- En relación a la representación de datos en memoria, en 64 bits, los punteros y los datos de tipo *long* ocupan 64 bits. El resto de tipos mantiene el mismo tamaño que en la línea de procesadores de 32 bits (int: 4 bytes, short: 2 bytes, …).
- Todas las instrucciones de 32 bits pueden utilizarse ahora con operandos de 64 bits. En ensamblador, el sufijo de instrucción para operandos de 64 bits es “q”.
- Los registros de 32 bits se extienden a 64. Para hacer referencia a ellos desde un programa en lenguaje ensamblador, hay que sustituir la letra inicial “e” por “r” (%rax, %rbx, …). Los registros de 32 bits de la IA32 se corresponden con los 32 bits de menor peso de estos nuevos registros.
- El byte bajo de los registros %rsi, %rdi, %rsp y %rbp es accesible. Desde el lenguaje ensamblador, el acceso a estos registros se realiza a través del nombre del registro de 16 bits finalizado con la letra “l” (%sil, %dil, …).
- Aparecen 8 nuevos registros de propósito general de 64 bits (%r8, %r9, …, %r15). Es posible acceder a los 4, 2 o al último byte de estos registros incluyendo en su nombre el sufijo d, w o b (%r8d – 4 bytes, %r8w – 2 bytes, %r8b – 1 byte).
- Aparecen nuevas instrucciones. Entre ellas, la instrucción que usaremos para el desarrollo de esta práctica es *movsx*. Esta instrucción permite representar el contenido del registro de 8, 16 ó 32 bits indicado como operando fuente en un registro de 64 bits indicado como operando destino (p.e., *movsx %eax, %rax*).

Ejemplo de implementación en ensamblador x86-64 de una función de C/C++

Dentro del fichero *imageprocess.cpp*, se ha incluido la implementación de la primera función (*suavizado*) a modo de ejemplo. Como ya se ha comentado anteriormente, la implementación de las restantes es objeto de esta práctica.

La función *suavizado* incluye 6 parámetros con la siguiente descripción:

- ***imagenO***: imagen origen de la operación. Se trata de un array de *H* filas y *W* columnas en el que cada elemento (píxel) contiene un valor entre 0 y 255 (unsigned char). El parámetro contiene la dirección inicial de dicho arrary.
- ***imagenD***: imagen donde debe almacenarse el resultado de la operación. Al igual que el parámetro anterior, es un array de *H* filas y *W* columnas en el que cada elemento contiene un valor entre 0 y 255 (unsigned char). Igualmente, el parámetro contiene la dirección inicial de dicho arrary.
- ***W***: número de columnas de *imagenO* e *imagenD*.
- ***H***: número de filas de *imagenO* e *imagenD*.
- ***kernel***: matriz utilizada para hacer la operación de suavizado, de acuerdo con la descripción indicada en la sección "*Funcionamiento de la aplicación*". El tamaño del array es de (2*N+1)x(2*N+1) y cada elemento es un entero de 32 bits. El parámetro marca la dirección de inicio del array.
- ***N***: parámetro de tipo entero que indica el tamaño del kernel de la operación según lo indicado anteriormente.

Cada uno de estos parámetros están indicados como operandos de entrada del bloque de código en ensamblador (segunda lista indicada al final del bloque *asm* precedida de ":"). El acceso a ellos desde el bloque de código en ensamblador debe hacer utilizando referencias (%0,%1, …) en lugar del nombre del parámetro. Concretamente, por el orden de definición de operandos del bloque, la relación entre parámetros y referencias es la siguiente: %0 = imagenO ; %1 = imagenD ; %2 = W ; %3 = H ; %4 = kernel ; %5 = N.

La implementación del código ensamblador se basa en el pseudocódigo proporcionado para el desarrollo de la práctica. La relación entre el pseudocódigo y la implementación en ensamblador se muestra en la siguiente tabla. En dicha tabla, se han marcado con colores diferentes los distintos bloques de código para ayudar a su interpretación. Como puede observarse, cada uno de los datos no asociados a parámetros que aparecen en el pseudocódigo (p.e., *dirImagenO, dirImagenD, y, x,* …) están representados a través de registros en la implementación en ensamblador.

Pseudocódigo	**Implementación en ensamblador**
dirImagenO = imagenO;	"mov %0, %%rsi;" *// %rsi es dirImagenO*
dirImagenD = imagenD;	"mov %1, %%rdi;" *// %rdi es dirImagenD*
Para(y=0; y<H; y++)	"mov $0, %%r8d;" *// %r8d es y*
{	"bSuavizadoY:"
Para(x=0; x<W; x++)	"mov $0, %%r9d;" *//%r9d es x*
{	"bSuavizadoX:"
dirKernel = kernel;	"mov %4, %%rbx;" *// %rbx es kernel*
acum = 0;	"mov $0, %%eax;" *// %eax es acum*
totalK = 0;	"mov $0, %%ecx;" *// %ecx es totalK*
Para(j=-N; j<=N; j++)	"mov %5, %%r10d;"
{	"neg %%r10d;" *// %r10d es j*
Para(i=-N; i<=N; i++)	"bSuavizadoKernelJ:"
{	"mov %5, %%r11d;"
Si((x+i)>=0 y (x+i)<W y (y+j)>=0 y (y+j)<H)	"neg %%r11d;" *// %r11d es i*

```
                {
                    offsetPixel = (y + j) * W + (x + i);
                    acum += [dirImagenO+offsetPixel]*[dirKernel];
                    totalK += [dirKernel];
                }
                dirKernel += 4;
            }
        }
        offsetPixel = y*W + x;
        [dirImagenD + offsetPixel] = acum / totalK;
    }
}
```

```
            "bSuavizadoKernelI:"
                "mov %%r8d, %%r12d;"
                "add %%r10d, %%r12d;"
                "mov %%r9d, %%r13d;"
                "add %%r11d, %%r13d;"
                "cmp $0, %%r12d;"
                "jl sigKernelI;"
                "cmp %3, %%r12d;"
                "jge sigKernelI;"
                "cmp $0, %%r13d;"
                "jl sigKernelI;"
                "cmp %2, %%r13d;"
                "jge sigKernelI;"
                "mov %2, %%r14d;"
                "imul %%r12d, %%r14d;"
                "add %%r13d, %%r14d;"
                "movsx %%r14d, %%r14;"
                "movzbl (%%rsi, %%r14), %%edx;"
                "imull (%%rbx), %%edx;"
                "add %%edx, %%eax;"
                "add (%%rbx), %%ecx;"
            "sigKernelI:"
                "add $4, %%rbx;"
                "inc %%r11d;"
                "cmp %5, %%r11d;"
                "jle bSuavizadoKernelI;"
            "sigKernelJ:"
                "inc %%r10d;"
                "cmp %5, %%r10d;"
                "jle bSuavizadoKernelJ;"
            "mov $0, %%edx;"
            "div %%ecx;" // acum/totalK
            "mov %%r8d, %%r14d;"
            "imul %2, %%r14d;"
            "add %%r9d, %%r14d;"
            "movsx %%r14d, %%r14;"
            "mov %%al, (%%rdi, %%r14);"
            "inc %%r9d;"
            "cmp %2, %%r9d;"
            "jl bSuavizadoX;"
        "inc %%r8d;"
        "cmp %3, %%r8d;"
        "jl bSuavizadoY;"
```

Esta implementación en ensamblador está ya incluida en la función *suavizado* de *imageprocess.cpp*. Al final del bloque *asm*, en la tercera lista precedida de ":" se han indicado todos los registros utilizados en la implementación. Es importante que en esta lista se especifiquen todos los registro usados para evitar que el compilador los utilice con otra funcionalidad. Si no se indican, puede ser motivo de error durante la ejecución del programa. Además de los registros, se incluye la especificación "memory" para indicar que la memoria es modificada durante la ejecución del bloque de código.

Especificación de los objetivos de la práctica

El principal objetivo de esta práctica es completar el código de la aplicación descrita para que su funcionamiento sea el que se detalla en la sección anterior. Para ello, se deberán implementar, en lenguaje ensamblador, los procedimientos vacíos del módulo "imageprocess.*cpp*". Para cada uno de ellos, se proporciona la estructura inicial del bloque ensamblador, en la que se ha incluido la definición de operandos que afectan a la implementación. La lista de registros utilizados incluye únicamente la palabra "memory", ya que en todos los casos la memoria es modificada. La inclusión de registros dentro de esta lista dependerá de la implementación que se desarrolle en cada caso, por lo que, será necesario completarla para cada uno de los procedimientos.

Se describe a continuación, con más detalle, la especificación de cada uno de los procedimientos a completar:

- *void* ***umbralizar****(unsigned char * **imagenO**, unsigned char * **imagenD**, unsigned char **umbral**, int **W**, int **H**)*: realiza la operación de umbralización descrita en la sección "*Funcionamiento de la aplicación*". Los parámetros del procedimiento tienen la siguiente descripción:
  - *imagenO*: imagen origen de la operación. El parámetro contiene la dirección de inicio de un array de *H* filas y *W* columnas donde cada elemento (píxel) contiene un *unsigned char* que representa el nivel de gris del píxel (valor comprendido entre 0 y 255).
  - *imagenD* : imagen destino de la operación con la misma estructura que el parámetro anterior.
  - *umbral*: valor umbral utilizado durante la operación. Está representado como unsigned char para poder especificar valores en el rango entre 0 y 255.
  - *W*: número de columnas de *imagenO* e *imagenD*.
  - *H*: número de filas de *imagenO* e *imagenD*.
- *void* ***calculoGradientes****(unsigned char* **imagen**, Gradient* **gradientes**, int **W**, int **H**):* realiza la operación de cálculo de gradientes (primera fase de la detección de bordes) descrita en la sección "*Funcionamiento de la aplicación*". Los parámetros del procedimiento tienen la siguiente descripción:
  - *imagen*: imagen origen de la operación. El parámetro contiene la dirección de inicio de un array de *H* filas y *W* columnas donde cada elemento (píxel) contiene un *unsigned char* que representa el nivel de gris del píxel (valor comprendido entre 0 y 255).
  - *gradientes*: array bidimensional donde se almacenan los gradientes horizontal y vertical de cada punto. El número de filas y columnas es el mismo que para el parámetro anterior. Cada elemento es una estructura de tipo *Gradient*, que almacena, en el atributo *dx*, el gradiente horizontal y, en el atributo *dy*, el vertical. El contenido del parámetro es la dirección de inicio de este array.
  - *W*: número de columnas de *imagen* y *gradientes*.
  - *H*: número de filas de *imagen* y *gradientes*.
- *void* ***supresionNoMaximo****(int * **gradNorm**, unsigned char * **gradDir**, unsigned char* **imagenD**, int **W**, int **H**):* realiza la operación de supresión del no máximo (segunda fase de la detección de bordes) descrita en la sección "*Funcionamiento de la aplicación*". Los parámetros del procedimiento tienen la siguiente descripción:

- *gradNorm*: array bidimensional de *H* filas y *W* columnas que contiene la norma del gradiente de cada píxel (*sqrt(dx*dx+dy*dy)*). Esta norma puede tener un valor superior a 255, por lo que cada elemento está definido de tipo *int*. El parámetro contiene la dirección de inicio del array.
- *gradDir*: array bidimensional de *H* filas y *W* columnas que contiene la dirección del gradiente de cada píxel. Cada elemento está representado con tipo *unsigned char*, almacenando un valor entre 0 y 3. El parámetro contiene la dirección de inicio del array.
- *imagenD*: imagen donde se almacena el resultado de la operación. Se trata de un array de tipo unsigned char (valores entre 0 y 255) en el que se almacena el resultado de la operación. El parámetro contiene la dirección inicial del array.
- *W*: número de columnas de los 3 arrays anteriores.
- *H*: número de filas de los 3 arrays anteriores.

- *void* ***dobleUmbralizacion****(unsigned char** ***imagenO****, unsigned char** ***imagenD****, unsigned char* ***umbralMin****, unsigned char* ***umbralMax****, int* ***W****, int* ***H****)*: realiza la operación de doble umbralización (tercera fase de la detección de bordes) descrita en la sección "*Funcionamiento de la aplicación*". Los parámetros del procedimiento tienen la siguiente descripción:
  - *imagenO*: imagen origen de la operación. El parámetro contiene la dirección de inicio de un array de *H* filas y *W* columnas donde cada elemento (píxel) contiene un *unsigned char* que representa el nivel de gris del píxel (valor comprendido entre 0 y 255).
  - *imagenD* : imagen destino de la operación con la misma estructura que el parámetro anterior.
  - *umbralMin*: umbral mínimo utilizado durante la operación. Está representado como unsigned char para poder especificar valores en el rango entre 0 y 255.
  - *umbralMax*: umbral máximo utilizado durante la operación. Está representado como unsigned char para poder especificar valores en el rango entre 0 y 255.
  - *W*: número de columnas de *imagenO* e *imagenD*.
  - *H*: número de filas de *imagenO* e *imagenD*.
- *int* ***detectaContorno****(unsigned char * **imagen****, unsigned char * **visitados****, int* ***ini_x****, int* ***ini_y****, Point* ***direcciones****[8], Point* ***contorno****[5000], int* ***W****, int* ***H****)*: realiza la operación de detección de contornos descrita en la sección "*Funcionamiento de la aplicación*". Los parámetros del procedimiento tienen la siguiente descripción:
  - *imagen*: imagen origen de la operación. El parámetro contiene la dirección de incio de un array de *H* filas y *W* columnas donde cada elemento (píxel) contiene un *unsigned char* que representa el nivel de gris del píxel (valor comprendido entre 0 y 255).
  - *visitados*: imagen utilizada para marcar los puntos visitados. Esta imagen tiene la misma estructura que el parámetro anterior.
  - *ini_x*: coordenada *x* del primer punto del contorno.
  - *ini_y*: coordenada *y* del primer punto del contorno.
  - *direcciones*: array de 8 elementos que contiene las posiciones relativas de los 8 vecinos de cada píxel. Cada elemento está definido de tipo *Point* para poder almacenar la posición relativa en columna (valor del atributo *x*) y en fila (valor del atributo *y*). El parámetro contiene la dirección de inicio del array.
  - *contorno*: array de puntos donde se deben almacenar las coordenadas de los píxels del contorno detectado. El número máximo de elementos de este array es

5000. Cada elemento es una estructura de tipo *Point* donde se almacenan las coordenadas *x* e *y* del punto. El parámetro contiene la dirección de inicio del array.

- *W*: número de columnas de *imagen* y *visitados*.
- *H*: número de filas de *imagen* y *visitados*.

Además de estos parámetros, la variable local *nPuntos* es utilizada como operando de salida del bloque de código ensamblador para poder almacenar el número de puntos del contorno detectado. Esta variable es retornada por el procedimiento.

- *int* ***contornoAPoligono****(Point* ***contorno****[5000], int* ***nPuntos****, Point* ***poligono****[5000], float* ***distancia****)*: realiza la operación de aproximación a polígonos descrita en la sección "*Funcionamiento de la aplicación*". Los parámetros del procedimiento tienen la siguiente descripción:
  - *contorno*: array de puntos donde se encuentran almacenados los puntos de un contorno. Cada elemento contiene las coordenadas *x* e *y* de un punto a través de una estructura de tipo *Point*. El parámetro contiene la dirección de inicio del array.
  - *nPuntos*: número de puntos del contorno.
  - *poligono*: array de puntos donde deben almacenarse las coordenadas de los puntos del polígono generado. Cada elemento es una estructura de tipo *Point* que almacena las coordenada *x* e *y* del punto. El parámetro contiene la dirección de inicio del array.
  - *distancia*: distancia umbral utilizada para dividir el contorno durante la aplicación de la operación.

  Además de estos parámetros, la variable local *nP* es utilizada como operando de salida del bloque de código ensamblador para poder almacenar el número de puntos del polígono. Esta variable es retornada por el procedimiento.

Es obligatorio implementar los procedimientos *umbralizar*, *calculoGradientes*, *supresionNoMaximo*, *dobleUmbralizacion* y *detectaContorno*. La nota máxima de la parte obligatoria será de Notable(8). La implementación del último procedimiento (*contornoAPoligono*) será opcional. Está implementación adicional podrá suponer hasta 2 puntos más en la nota de la práctica.

Nota: la práctica se realizará de manera individual.
Fecha de entrega de la práctica: 25/01/2024 (hasta las 14:00)
Entrega: la entrega se realizará a través de la subida al aula virtual de la asignatura de un archivo **.zip** que contenga el proyecto completo.