PROYECTO REDES MULTIMEDIA

IMPLEMENTACIÓN: EBCOT

1. Ficheros de ejemplo disponibles

Para la implementación de EBCOT dispones del fichero *lena.bl1*, que es un ejemplo de un bloque de 64x64 a codificar.

2. Funciones disponibles

En el fichero *imagenES.cpp* se disponen de las siguientes funciones, tanto para la carga y grabación de ficheros, como para la reserva de bloques.

2.1. Reserva de bloques

En concreto, para la reserva de memoria para un bloque tenemos la función:

```
void ReservaPlano(int ancho, int alto, int ***Plano);
```

Un ejemplo de uso sería el siguiente:

```
int **BloqueEj;
ReservaPlano(64, 64, &BloqueEj);
```

Recuerda que también puede haber bloques que sean menores de 64x64 (por ejemplo, para subbandas más pequeñas).

2.2. Carga y grabación de bloques

Para la carga de un bloque se usa la función

```
int CargaBloque(char *Nombre, int *ancho, int *alto, int *nivel, int *subbanda, int **Bloque);
```

Esta función devuelve los parámetros de ancho, alto, nivel y subbanda que se leen de la cabecera.

En concreto, los parámetros de esta función son los que siguen:

a) el primer parámetro de esta función es el nombre del fichero a leer,
b) el segundo y tercer parámetro indican las dimensiones del bloque,
c) el cuarto parámetro indica el nivel de descomposición al que pertenece el bloque (de 0 a N-1, siendo N el numero de descomposiciones de la DWT),
d) el quinto parámetro indica el tipo de subbanda a la que pertenece (puede ser HH, LH, HL, LL, cuyos valores están declarados en ImagenES.h),
e) y por ultimo se pasa un puntero al bloque donde queremos almacenar la información que se lee.

Así, un ejemplo de uso sería:

```
CargaBloque("ejemplo.bl1", &anchoBl, &altoBl, &nivel, &sub, BloqueEj);
```

Para la grabación se usará la función

```
int GuardaBloque(char *Nombre, int ancho, int alto, int nivel,
int subbanda, int **Bloque);
```

Cada bloque tendrá una pequeña cabecera en la que se guardan el ancho, alto, nivel y subbanda del bloque, usando un byte para cada uno de estos parámetros.

Así, un ejemplo de uso sería:

```
GuardaBloque("ejemplo.bl1", 64, 64, 1, HH, BloqueEj);
```

2.3. Tratamiento de ficheros de bits

El módulo *FichBits.cpp* es exactamente igual al usado en prácticas, y se empleará para leer y escribir la secuencia de bits que genera el codificador EBCOT para cada uno de los bloques.

El fichero que contiene los bits que EBCOT ha generado tiene como extension .biX (por ejemplo, *ej.bi1*).

Para su escritura se usan las funciones:

```
InicializaEscritura("ej.bi1")
EscribeBit(X)
FinalizaEscritura().
```

También se disponen de las funciones equivalentes para la lectura.

Por otro lado, por cada uno de los bits se debe guardar el contexto a utilizar. Usamos para ello un fichero de extensión .ctx. Proponemos usar un fichero binario estándar, usando un byte por cada contexto de cada bit guardado. Un ejemplo de uso para escritura de contextos seria el siguiente

```
FILE *FCtxt=fopen("ej.ct1","wb");
putc(X1,FCtxt);  // X1: contexto del primer bit
putc(X2,FCtxt);  // X2: contexto del segundo bit
 ....
fclose(FCtxt);
```

Su lectura se puede hacer de forma equivalente (habría que cambiar "wb" por "rb" y usar `getc(FCtxt)` en lugar de `putc`)

Por ultimo, para la formación del bitstream podría ser necesario conocer la longitud de cada pasada de EBCOT (numero de coeficientes codificados por pasada). Esta información, junto con una cabecera con información sobre el bloque (similar a la generada por GuardaBloque(...)) e información sobre cual es la primera capa de bits codificada, se guarda en un fichero de extension .lpx. Nuevamente usaremos ficheros estándar, pero en este caso usaremos dos bytes para almacenar las longitudes (ya que el rango de este valor supera 255). Por ejemplo:

```
FILE *FLp=fopen("ej.lp1","wb");
putc(ancho,FLp);
putc(alto,FLp);
putc(nivel,FLp);
putc(subbanda,FLp);
putc(primer_plano,FLp);  // primer capa de bits codificada
                    //(la primera con coeficientes signficativos)
putc(L1/256,FLp); // L1: longitud de la primera pasada (MSB)
putc(L1%256,FLp); // L1: longitud de la primera pasada (LSB)
putc(L2/256,FLp); // L2: longitud de la segunda pasada (MSB)
putc(L2%256,FLp); // L2: longitud de la segunda pasada (LSB)
  ....
fclose(FLp);
```

Observa que, en la lectura, se pueden recuperar los valores de la siguiente forma:

```
  ....

L1high= getc(FLp); L1low=getc(FLp); L1=L1high*256+L1low;
L2high= getc(FLp); L2low=getc(FLp); L2=L2high*256+L2low;
  ....
```

3. Lectura a nivel de bit

Es importante recordar que para acceder a los distintos bits de un entero en C, se puede utilizar los siguientes operadores a nivel de bit (*bitwise*)

- "a | b" para aplicar un OR bit a bit entre a y b
- "a & b" para aplicar un AND bit a bit entre a y b
- "~a" para aplicar un NOT bit a bit de a
- "a>>x" para desplazar los bits del entero a, hacia la derecha x bits
- "a<<x" para desplazar los bits del entero a, hacia la izquierda x bits

De esta manera, si queremos comprobar, por ejemplo, si el quinto bit de una variable "a" es 0, podemos hacerlo utilizando máscaras de la siguiente forma

```
if ( (a&(1<<5-1)==0) printf("es cero"); else ("es uno")
```

4. Ejemplo de codificación

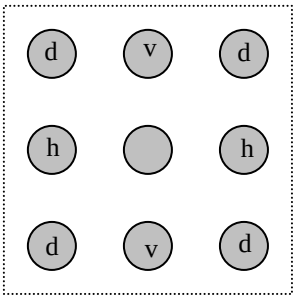
A modo de ejemplo, destacar que al codificar el bloque que se proporciona como ejemplo, *lena.bl1*, se codifican cinco planos de bits, de forma que en cada uno de los planos de bits, el número de coeficientes que se codifican en cada pasada es el siguiente:

Plano	**Pasada**	**Número de coeficientes**
4	*Propagación*	0
	Refinamiento	0
	Clean-up	4096
	Total	*4096*
3	*Propagación*	8
	Refinamiento	1
	Clean-up	4087
	Total	*4096*

2	*Propagación*	32
	Refinamiento	4
	Clean-up	4060
	Total	*4096*
1	*Propagación*	1610
	Refinamiento	312
	Clean-up	2174
	Total	*4096*
0	*Propagación*	2442
	Refinamiento	1462
	Clean-up	192
	Total	*4096*

5. Uso de contextos

Para una implementación eficiente de EBCOT, se deben utilizar distintos contextos en función de la situación y los vecinos del coeficiente cuyo bit se va a almacenar. En concreto, vamos a considerar los siguientes vecinos horizontales, verticales, y diagonales, de la siguiente forma:



5.1. Contextos en los pasos de propagación y clean-up

En los pasos de propagación y clean-up, a partir de la significancia de estos vecinos, podemos obtener los distintos contextos con los que codificar cada bit.

Contextos para subbandas LL y LH			Etiqueta de contexto
∑ h	∑ v	∑ d	
2	x	x	8
1	>=1	x	7
1	0	>=1	6
1	0	0	5
0	2	x	4
0	1	x	3
0	0	>=2	2
0	0	1	1
0	0	0	0

Contextos para HL			Etiqueta de contexto
∑ h	∑ v	∑ d	
x	2	x	8

>=1	1	x	7
0	1	>=1	6
0	1	0	5
2	0	x	4
1	0	x	3
0	0	>=2	2
0	0	1	1
0	0	0	0

Contextos para HH		Etiqueta de contexto
∑ (h+v)	∑ d	
x	>=3	8
>=1	2	7
0	2	6
>=2	1	5
1	1	4
0	1	3
>=2	0	2
1	0	1
0	0	0

De esta forma, al codificar una subbanda LH, si son significativos nuestro vecino derecho, el vecino de arriba, y los dos que se encuentran en las diagonales superiores, tendremos una significancia de (1, 1, 2), con lo que usaremos el contexto 7.

5.2. Contextos en los pasos de refinamiento

En el caso de codificar un bit en la etapa de refinamiento, el contexto que se le puede asociar es distinto a las otras dos etapas del algoritmo. En este caso sólo hay 3 contextos posibles, que dependen de si es la primera vez que se refina el coeficiente y de si tiene vecinos significativos. En concreto, si es la primera vez que se refina el coeficiente (es decir, se hizo significativo justo en el plano de bit anterior) se pueden aplicar dos contextos: el 15 si tiene algún vecino significativo o el 14 si no lo tiene. En caso de que el coeficiente se haya refinado ya otras veces se aplica el contexto número 16.

5.3. Contextos de signos

Al codificar el signo también se aplica un contexto al bit de signo. En este caso el contexto depende de si el coeficiente tiene vecinos significativos en vertical u horizontal, y del signo de esos vecinos. Concretamente usamos dos variables (H y V) que tomarán los siguientes valores según sean los vecinos (los horizontales o los verticales):

- Un +1 en la variable indica que ambos vecinos son significativos y tienen signo positivo.
- Un -1 en la variable supondrá que los dos vecinos son significativos y negativos.
- En otro caso, se usa el valor de 0 en la variable.

A continuación dependiendo del valor de estas variables se aplican los contextos correspondientes siguiendo las reglas definidas en la siguiente tabla:

Contexto	13	12	11	10	9	10	11	12	13
H	1	1	1	0	0	0	-1	-1	-1
V	1	0	-1	1	0	-1	1	0	-1

6. Algunos detalles para la implementación

Aunque ya se proporciona un bloque de ejemplo a partir del que se puede trabajar, en algunos casos podría ser interesante disponer de más bloques, o incluso realizar una prueba en un entorno completo (antes de la prueba en equipo). Con esta finalidad, también tenéis disponible un ejemplo completo, en el que a partir de dos programas y una versión reducida de la imagen de *lena* (en tonos de grises), se pueden generar todos los bloques y codificarlos con una versión ya implementada de EBCOT. El ejecutable de EBCOT recibe como parámetro el nombre de los bloques (sin extensión) y la cantidad de bloques a procesar, de forma que se pueden comprimir varios bloques con una sola llamada. Al implementar vuestro codificador/decodificador, podríais recibir los parámetros por comando de forma parecida. Junto al ejemplo, tenéis disponible un fichero .pdf con una descripción del mismo.

Algunos detalles que podéis contemplar en la decodificación son los siguientes:

1) El decodificador podría encontrarse con bloques no completos, (es decir, en los que las últimas pasadas podrían no haberse leído al aplicar escalabilidad SNR). Así un parámetro que sería interesante incluir en el decodificador es el número de planos de bits de menor peso que no vamos a leer (por ejemplo, si no queremos leer los dos planos de menor peso, este parámetro sería 2, si queremos leer todo el bloque sería 0).

2) Una buena implementación del decodificador no necesitaría leer los ficheros de contextos y los planos de bit (excepto para conocer el tamaño y situación de los bloques), ya que estos valores los puede calcular de manera dinámica, igual que hace el codificador.

En vuestra implementación, no es obligatorio tener en cuenta estas consideraciones, pero sí mejorarían la calidad del trabajo si son seguidas.