Ejercicio obligatorio 4

Fecha de entrega: Domingo 6 de noviembre

Introducción

Paletas de colores

Las paletas de colores son una técnica que se utiliza para almacenar y generar imágenes economizando recursos. La idea del método consiste en representar las imágenes como índices a tablas de colores.

Una paleta no es más que una tabla con una cantidad N de registros. Cada uno de los registros es un color en determinado formato. Dado un índice de esta tabla se obtiene el color correspondiente.

Teniendo esta tabla en vez de representarse cada pixel de la imagen como un color lo que se almacena es un índice. Es decir, la imagen será una matriz de índices de la tabla.

Mediante esta técnica se puede reducir notablemente la cantidad de espacio necesario para almacenar una imagen. Es común en juegos clásicos que las imágenes se almacenen en 4, 3 o incluso 2 bits, y que se apliquen paletas de colores sobre ellos. Además es muy común representar diferentes objetos con la misma imagen pero sólo cambiando la paleta (por ejemplo, en el Super Mario Bros. los árbustos y las nubes utilizan exactamente la misma imagen pero con diferente paleta).

Formato PPM

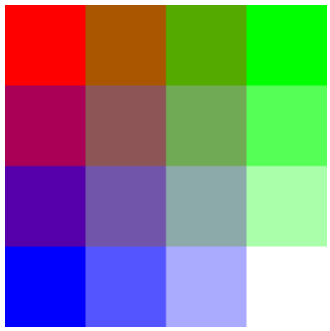
El formato Portable Pixel Map (PPM) es el formato que ofrece el paquete Netpbm para la representación de imágenes a color en un archivo sencillo.

El formato es similar al PGM ya analizado con la diferencia de que ahora por cada pixel en vez de tener un único valor de grises tendremos tres valores, uno para cada componente RGB del color.

Por ejemplo el siguiente archivo

```
P3
# Este es un ejemplo de PPM
4 4
# Asumimos que el máximo es siempre 255
255
#   Rojo                                  Verde
255   0   0   170  85   0    85 170   0     0 255   0
170   0  85   141  85  85   113 170  85    85 255  85
 85   0 170   113  85 170   141 170 170   170 255 170
  0   0 255    85  85 255   170 170 255   255 255 255
#   Azul                                    Blanco
```

genera la siguiente imagen:



Resumiendo: El encabezado es P3, hay 3 valores numéricos por cada pixel.

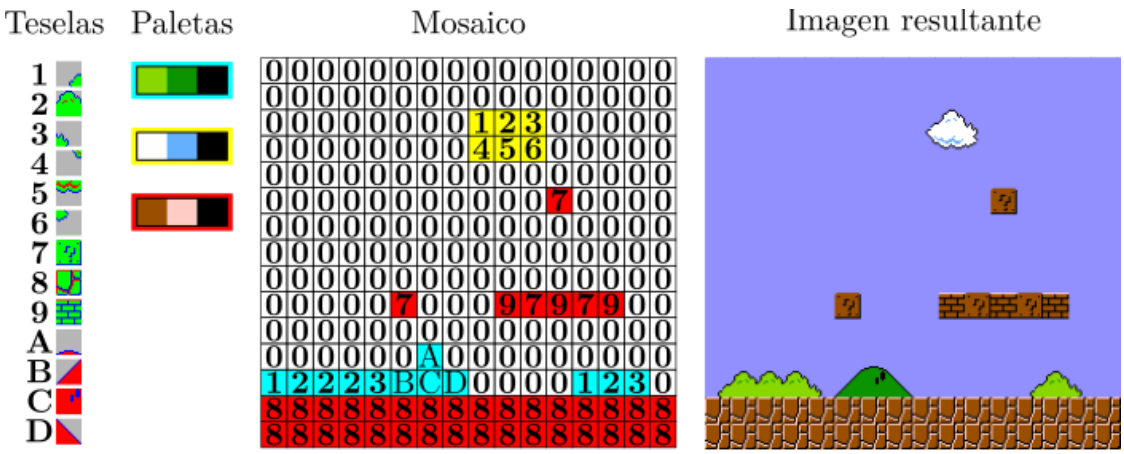
Mosaicos y teselas

Cuando hay que generar gráficas complejas y hay restricciones de memoria una técnica habitual consiste en almacenar pequeñas imágenes de un tamaño fijo, llamadas teselas, y componer las gráficas completas uniendo estas teselas para formar un mosaico.

Por ejemplo (y simplificando mucho), la consola NES (Family Game) tiene una resolución de pantalla de 256x240 pixeles y un fondo se puede crear con hasta 256 teselas de 8x8 pixeles y 2 bits de color.

El Super Mario Bros. utiliza teselas de 16x16 pixeles, por lo que una pantalla se llena con 16 teselas en el ancho y 15 teselas en el alto. Estas teselas son elegibles de una lista que tiene 64 teselas posibles donde de los 4 colores que pueden seleccionarse con 2 bits el primero corresponde al fondo transparente. Luego las paletas tendrán 3 colores, para las otras 3 combinaciones restantes.

La siguiente imagen muestra para una determinada pantalla del juego las teselas con las que se compone la imagen, sus paletas y el diseño del mosaico en 16x15 mostrando qué tesela y qué paleta se usa en cada caso:



Notar que, como se dijo anteriormente, los arbustos y las nubes utilizan las mismas teselas; mientras que las montañas y los arbustos utilizan la misma paleta.

Con estas técnicas (y algunos detalles que se simplificaron en este ejemplo) el juego completo, con toda su arte gráfica, entra en apenas 40KB de memoria. Para que se entienda, almacenar una imagen de 120x115 con 24 bits de color ocupa más que el juego completo con todos sus niveles, sus enemigos, su música, su programación, todo.

Pixeles

Hay múltiples maneras de almacenar pixeles según diferentes profundidades de color, del mismo modo que un pixel se puede interpretar de diferentes maneras según se considere una representación de colores RGB o el índice de una tabla. Es decir, en el caso de que se almacene un

valor de n bits eso podrá ser interpretado indistintamene como un color de n bits o como un índice entre 0 y $2^n - 1$ según el contexto.

De todas las formas posibles en este trabajo vamos a trabajar con dos formatos en particular: De 3 bits y de 12 bits.

El formato de pixeles de 3 bits será el siguiente:

```
2            0
+----+----+----+
| r0 | g0 | b0 |
+----+----+----+
```

en este formato se pueden representar 8 colores diferentes. Para convertir este color a RGB de 24 bits, que es lo que se usa en los dispositivos gráficos si el bit correspondiente al color está en 0 eso es un valor de 0, mientras que si el bit está puesto en 1 eso es un valor de 255.

Como ya se dijo, hay una dualidad, este formato de 3 bits puede representar tanto 8 colores puros o puede representar 8 valores en una paleta de colores en otra representación.

El formato de pixeles de 12 bits será el siguiente:

```
11                8                                       0
+----+----+----+----++----+----+----+----+----+----+----+----+
| r3 | r2 | r1 | r0 || g3 | g2 | g1 | g0 | b3 | b2 | b1 | b0 |
+----+----+----+----++----+----+----+----+----+----+----+----+
```

En este caso cada componente puede tomar un valor entre 0 y 15, lo que da 4096 colores diferentes o puede representar 4096 valores en una paleta de colores en otra representación. Para convertir cada una de estas componentes a representación de 24 bits se deben desplazar 4 bits a la izquierda, así el valor `0xf` (15) se convertirá en `0xf0` (240).

Formato de teselas 8x8 1bpp

Para representar teselas de 8x8 en 1 bit de color se suelen usar 8 bytes en secuencia. Los valores de los bits del primer byte corresponden a los valores de los pixeles de la primera fila y así para cada uno de los valores.

Por ejemplo, la secuencia:

```
secuencia_r[8] = {0xf7, 0xf7, 0xff, 0xff, 0xff, 0xf7, 0xf7, 0xf7};
```

se corresponde con la tesela:

```
pixeles_r[8][8] = {
    {1, 1, 1, 1, 0, 1, 1, 1},
    {1, 1, 1, 1, 0, 1, 1, 1},
    {1, 1, 1, 1, 1, 1, 1, 1},
    {1, 1, 1, 1, 1, 1, 1, 1},
    {1, 1, 1, 1, 1, 1, 1, 1},
    {1, 1, 1, 1, 0, 1, 1, 1},
    {1, 1, 1, 1, 0, 1, 1, 1},
    {1, 1, 1, 1, 0, 1, 1, 1}
};
```

Mientras que la secuencia:

```
secuencia_g[8] = {0x00, 0x62, 0x62, 0x7e, 0x62, 0x62, 0x62, 0x00};
```

se corresponde con la tesela:

```
pixeles_g[8][8] = {
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 1, 1, 0, 0, 0, 1, 0},
    {0, 1, 1, 0, 0, 0, 1, 0},
    {0, 1, 1, 1, 1, 1, 1, 0},
    {0, 1, 1, 0, 0, 0, 1, 0},
    {0, 1, 1, 0, 0, 0, 1, 0},
    {0, 1, 1, 0, 0, 0, 1, 0},
    {0, 0, 0, 0, 0, 0, 0, 0}
};
```

La arquitectura System 16 de Sega, por ejemplo, utiliza ROMs de 32KB para almacenar 4096 teselas de 8x8 pixeles de 1 bit. En estas memorias hay un continuo de bytes que define, de a paquetes de 8, los pixeles de cada una de estas teselas.

Ahora bien, para generar imágenes con más de 1 bit de color en vez de complejizar el formato anterior, lo más sencillo es implementar un interlineado. En vez de proponer un nuevo formato que permita teselas con más bits la estrategia es que cada componente de color se lea de una ROM diferente, con sus 4096 teselas y usar cada una de las ROMs para componer el color.

Por ejemplo, para los volcados de memoria `6841.rom`, `6842.rom` y `6843.rom` de Sega desde la posición 576 a la 583 de cada uno de los archivos se tiene la siguiente secuencia:

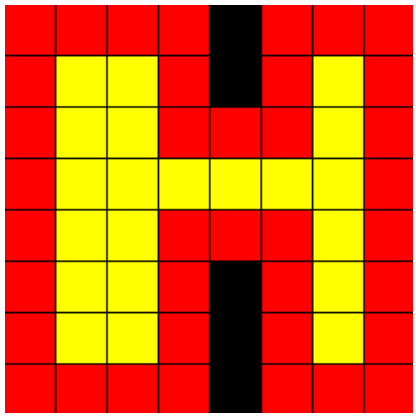
```
secuencia_6841[8] = {0xf7, 0xf7, 0xff, 0xff, 0xff, 0xf7, 0xf7, 0xf7};
secuencia_6842[8] = {0x00, 0x62, 0x62, 0x7e, 0x62, 0x62, 0x62, 0x00};
secuencia_6843[8] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
```

donde para una tesela de 3 bits el primer archivo va a aportar el bit de rojo, el segundo el de verde y el tercero el de azul. Entonces, combinando los 3 bits en una única tesela el resultado será:

```
pixeles_rgb[8][8] = {
    {4, 4, 4, 4, 0, 4, 4, 4},
    {4, 6, 6, 4, 0, 4, 6, 4},
    {4, 6, 6, 4, 4, 4, 6, 4},
    {4, 6, 6, 6, 6, 6, 6, 4},
    {4, 6, 6, 4, 4, 4, 6, 4},
    {4, 6, 6, 4, 0, 4, 6, 4},
    {4, 6, 6, 4, 0, 4, 6, 4},
    {4, 4, 4, 4, 0, 4, 4, 4}
};
```

(donde 4 es el valor binario `0b100` y 6 el valor binario `0b110`).

Si imprimimos esta imagen según la convención de pixeles de 3 bits dada el resultado será:



¿Dijimos que eso estaba en la posición 576 de los archivos? Si cada tesela ocupa 8 bytes, entonces esa es la tesela número 72 almacenada ahí. No debería ser sorpresa para nadie que convenientemente `'H' == 72`.

Trabajo

`pixel_t`

En el nuevo planteo el `pixel_t` sirve para almacenar píxeles tanto en formato de 3 bits como en formato de 12 bits. Dado que el tipo es uno solo, se utilizará:

```
typedef uint16_t pixel_t;
```

en ambos casos.

Implementar una función `pixel_t pixel3_crear(bool r, bool g, bool b);` que dados valores de `r`, `g` y `b` (0 o 1) empaquete eso según el formato de pixel de 3 bits dado.

Implementar una función `void pixel3_a_rgb(pixel_t pixel3, uint8_t *r, uint8_t *g, uint8_t *b);` que dado un pixel de 3 bits devuelva las componentes de RGB de 24 bits según la conversión ya explicada.

Implementar una función `pixel_t pixel12_crear(uint8_t r, uint8_t g, uint8_t b);` que dados valores de `r`, `g` y `b` (entre 0 y 15) empaquete eso según el formato de pixel de 12 bits.

Implementar una función `void pixel12_a_rgb(pixel_t pixel12, uint8_t *r, uint8_t *g, uint8_t *b);` que dado un pixel de 12 bits devuelva las componentes de RGB de 24 bits.

> **Nota**
>
> Cabe destacar que el `pixel_t` no es un TDA en el sentido estricto, su representación interna está dada por especificación y es conocida. Es decir para incializar un pixel de 12 bits rojo puede usarse tanto `pixel12_crear(15, 0, 0)` como `0xf00`, son construcciones equivalentes, una de más bajo nivel que la otra, pero que generan el mismo resultado.

TDA Imagen

Se tenía que decir y se dijo: `imagen_t` siempre fue un TDA, aun cuando no supiéramos de TDAs.

Todas las funciones ya implementadas, con excepción de `_imagen_crear()` son ahora primitivas del TDA. Esta última función es sólo eso, una función auxiliar del TDA que no debería ser conocida desde afuera.

Getters y setters

Según este paradigma, a partir de ahora está prohibido acceder a la representación interna del tipo así que se pide implementar las siguientes primitivas:

- `size_t imagen_get_ancho(const imagen_t *im);` : Getter del ancho de la imagen.
- `size_t imagen_get_alto(const imagen_t *im);` : Getter del alto de la imagen.
- `pixel_t imagen_get_pixel(const imagen_t *im, size_t x, size_t y);` : Getter del pixel en la posición `x`, `y`.
- `bool imagen_set_pixel(const imagen_t *im, size_t x, size_t y, pixel_t p);` : Setter del pixel en la posición `x`, `y` con el valor `p`.

Manejo de paletas

Implementar la primitiva `void imagen_pegar_con_paleta(imagen_t *destino, const imagen_t *origen, int x, int y, const pixel_t paleta[]);` que se comporta igual que la primitiva `imagen_pegar()` ya desarrollada con la particularidad de que para cada pixel `p` de la imagen `origen` lo que se debe asignar en la imagen `destino` es el valor correspondiente a `paleta[p]`. En el caso de que `p` sea `0` se considera transparente, como hasta ahora.

Salida en PPM

Implementar la primitiva `void imagen_escribir_ppm(const imagen_t *im, FILE *fo, void (*pixel_a_rgb)(pixel_t, uint8_t *, uint8_t *, uint8_t *));` que reciba un archivo `fo` abierto en modo escritura y texto y escriba en él la imagen `im` en formato PPM. Dado que el TDA imagen no conoce el formato de los pixeles que la componen, se le pasa además el puntero a función `pixel_a_rgb` que es una función capaz de extraer las componentes RGB de 24 bits para cada pixel de la imagen.

Implementar la primitiva `bool imagen_guardar_ppm(const imagen_t *im, const char *fn, void (*pixel_a_rgb)(pixel_t, uint8_t *, uint8_t *, uint8_t *));` similar a la anterior, pero que en vez de recibir un archivo recibe la ruta del mismo. Se debe devolver `true` en caso de éxito.

Manejo de mosaicos y teselas

Implementar una función `bool leer_teselas(imagen_t *teselas[]);` que reciba un vector con 4096 imágenes de 8x8 ya inicializadas y cargue en ellas las teselas contenidas en los archivos `ARCHIVO_ROM_R`, `ARCHIVO_ROM_G` y `ARCHIVO_ROM_B`. Se debe devolver `true` si todo está bien.

> **Nota**
>
> Para resolver este problema hacerlo de forma secuencial, cargar primero un archivo, luego el siguiente y luego el último y componer los colores de la imagen mientras se recorre.
>
> Para este problema particular bien podrían recorrerse los tres archivos a la vez en paralelo, pero en el TP1 a aparecer una variante de este procedimiento que sí o sí tiene que ser resuelta secuencialmente y se van a simplificar las cosas si este ya se encara con esa lógica.

Implementar una función `imagen_t *generar_mosaico(imagen_t *teselas[], const pixel_t paleta[][8], size_t filas, size_t columnas, const uint16_t mosaico_teselas[filas][columnas], const uint8_t mosaico_paletas[filas][columnas]);` que dado el vector con las `teselas` y un vector la `paleta` reciba una tabla `mosaico_teselas` con los índices de cada tesela y otra tabla `mosaico_paletas` con los índices de la paleta a usar, y genere una imagen que tenga `columnas` x `filas` teselas.

Aplicación

Se proveen:

- Un `main.c` que ejecuta estas funciones.
- Un par de archivos `paleta.c` y `paleta.h` que contienen los vectores de paletas.
- Un par de archivos `fondo.c` y `fondo.h` que contienen los mosaicos para generar dos imágenes en base a las teselas.
- Las 3 ROMs `6841.rom`, `6842.rom` y `6843.rom` ya mencionadas.
- Todos los PPM que genera el `main.c` al ser ejecutado.

El paquete con todos estos archivos puede descargarse de acá: archivos_20222_ej4.tar.gz

El resultado principal de invocar al programa deberá ser la siguiente imagen:

Nota

Para este trabajo no es necesario modularizar el problema, pero se recomienda hacerlo para practicar modularización y Makefile.

En el caso de modularizar en archivos entregar tanto los archivos como el Makefile que compila el proyecto.

Entrega

Deberá entregarse el código fuente del programa desarrollado o los fuentes y el Makefile en caso de haber modularizado.

El programa debe:

1. Compilar correctamente con los flags:

```
-Wall -Werror -std=c99 -pedantic
```

2. validar que se generen archivos PPM idénticos a los provistos.

La entrega se realiza a través del sistema de entregas.

El ejercicio es de entrega individual.