Examen final 2020-09-14

95.11/75.02 - Algoritmos y Programación I - Curso Essaya

Objetivo

Implementar el TDA `img`, que representa una imagen digital de tamaño arbitrario.

Se dispone de los siguientes archivos:

- **Archivos provistos con código:**
  - `tipografia8x8.h`: Tipografia 8x8 (header)
  - `tipografia8x8.c`: Tipografia 8x8 (implementación)
  - `pruebas.c`: función `main` + pruebas
  - `Makefile`
- **Archivos a completar con código:**
  - `rgba.h`: TDA `rgba` (header)
  - `rgba.c`: TDA `rgba` (implementación)
  - `img.c`: TDA `img` (implementación)
  - `img.h`: TDA `img` (header)

La idea es completar los archivos indicados de forma tal que el programa funcione.

Al compilar con `make`, se genera el archivo ejecutable `img_pruebas`. El mismo, al ejecutarlo, efectúa una serie de pruebas para verificar el correcto funcionamiento del TDA `img`.

Las pruebas están divididas en 5 ejercicios. Es condición necesaria (pero no suficiente) para aprobar el examen que haya 3 ejercicios OK.

Configuración de los ejercicios

Los ejercicios no están todos habilitados por defecto (para que se pueda compilar el programa sin haber implementado todas las funciones del TDA). Para habilitar la compilación de un ejercicio, por ejemplo el ejercicio 1, descomentar en `pruebas.c` la línea que dice:

```
//#define EJERCICIO1_HABILITADO
```

es decir, que quede de la siguiente manera:

```
#define EJERCICIO1_HABILITADO
```

y volver a compilar el programa.

Salida del programa

Al ejecutar el programa (`./img_pruebas`), se ejecutan todos los ejercicios habilitados, y se imprime el resultado de cada uno (OK o FAIL), junto con la cantidad de ejercicios OK. Ejemplo:

```
$ ./img_pruebas
ejercicio 1: OK
ejercicio 2: OK
ejercicio 3: OK
Prueba fallida en pruebas.c:122: img_get_pixel(img, x, y) == rgba(x / 2, y / 2, x / 2, y)
```

```
ejercicio 4: FAIL
ejercicio 5: OK
Cantidad de ejercicios OK: 4
```

Recordar: es condición necesaria (pero no suficiente) para aprobar el examen que haya al menos 3 ejercicios OK.

Recordar: Correr el programa con `valgrind --leak-check=full` para mayor seguridad de que la implementación es correcta.

Descripción de los TDAs

- Una **imagen** (TDA `img`) se representa digitalmente mediante un conjunto de `ancho x alto` pixels.
- Cada **pixel** puede tomar un color determinado, que se representa mediante el TDA `rgba`.
- Cada pixel tiene una posición dada por un par de coordanadas `(x, y)`. Las coordenadas `(0, 0)` representan el pixel en la esquina superior izquierda, y `(ancho - 1, alto - 1)` es el pixel en la esquina inferior derecha.
- El TDA `rgba` a su vez tiene 4 **componentes**: Rojo (R), Verde (G), Azul (B) y Alpha (A).
- Cada **componente** puede tomar valores entre 0 y 255, siendo 0 la ausencia total del componente. De esta manera, por ejemplo, el pixel `(r=0, g=0, b=0, a=255)` representa el color negro, `(r=255, g=0, b=0, a=255)`, el color rojo y `(r=255, g=255, b=255, a=255)` el color blanco.
- La componente alpha tiene distintos usos según el contexto. En las pruebas tomará casi siempre el valor 255. A los efectos del código debe comportarse como si fuera un componente más, a menos que se indique lo contrario.

Descripción de los ejercicios

EJERCICIO 1 (funcionamiento básico) Prueba el funcionamiento básico de los TDAs. Sin este ejercicio funcionando probablemente no se pueda pasar ninguna de las otras pruebas.

Tipos a declarar:

- Declarar un `typedef` apropiado para `componente_t` y `rgba_t`. Ambos deben ser tipos enteros.
- Declarar un `typedef` apropiado para `img_t`.

Funciones a implementar:

- `rgba`: recibe las 4 componentes por separado (tipo `componente_t`) y devuelve el color (tipo `rgba_t`).
- `rgba_r`, `rgba_g`, `rgba_b`, `rgba_a`: Reciben un color y devuelven cada una de las componentes.
- `img_crear`: recibe el ancho y el alto (cantidad de columnas y filas de pixels, respectivamente), y crea una imagen con esas dimensiones. No inicializa el color de los pixels (es decir, se asume que una imagen recién creada contiene basura).
- `img_destruir`: Libera la memoria asociada con la imagen.
- `img_ancho`, `img_alto`: devuelven las dimensiones de una imagen
- `img_set_pixel`: Recibe un par de coordenadas `(x, y)` y un color (`rgba_t`), y asigna el color al pixel en la posición indicada. Si las coordenadas están fuera de rango, no hace nada.
- `img_get_pixel`: Recibe `(x, y)` y devuelve el color del pixel en la posición indicada. Si las coordenadas están fuera de rango, devuelve el color negro.
- `img_clonar`: Devuelve una copia exacta de la imagen.

EJERCICIO 2 (archivos) Prueba que la imagen pueda ser guardada en un archivo y luego leída.

Funciones a implementar:

- `img_escribir`: Recibe un archivo previamente abierto en modo binario, y escribe en él el contenido de la imagen. Devuelve un booleano indicando si la operación fue exitosa.
- `img_leer`: Recibe un archivo previamente abierto en modo binario, y a partir los datos leidos del mismo crea una imagen nueva y la devuelve. Devuelve `NULL` en caso de error.

EJERCICIO 3 (ordenar imágenes) Prueba que podamos ordenar un arreglo de imágenes por tamaño.

Definimos el **tamaño** de una imagen como la cantidad total de pixels que contiene, es decir, el resultado de `ancho × alto`.

Funciones a implementar:

- `img_ordenar_por_tamagno`: Recibe un arreglo de imágenes y la cantidad de elementos que contiene. Ordena el arreglo según el tamaño de las imágenes, de menor a mayor.

Sugerencia: para el ordenamiento se puede implementar alguno de los algoritmos que vimos en clase (cualquiera), o bien utilizar la función `qsort` de la librería estándar de C.

EJERCICIO 4 (sellar) Prueba que se pueda "sellar" una imagen sobre otra.

Funciones a implementar:

- `img_pintar`: Recibe un color, y pinta todos los pixels con ese color.
- `img_sellar`: Recibe una imagen de fondo (F), una imagen sello (S) y un par de coordenadas `(x, y)`. La imagen F se modifica "calcando" la imagen S en las coordenadas indicadas (tomando como origen siempre la esquina superior izquierda), como si F fuera un papel y S fuera un sello que se presiona sobre el papel.

  Los pixels de la imagen S que tengan `alpha = 0` se ignorarán, es decir que no se sellarán sobre F. Para cualquier pixel de S que tenga alpha distinto de 0, el color será asignado sobre F (las cuatro componentes, incluyendo alpha) en la coordenada correspondiente.

Los pixels de la imagen S que queden fuera del rango de F también se ignorarán. La imagen F no cambia de tamaño.

EJERCICIO 5 (texto) Prueba que se pueda crear una imagen a partir de una cadena de texto.

Funciones a implementar:

- `img_crear_texto`: Recibe una cadena, un color de fondo y un color de texto. Devuelve una imagen nueva con el texto ”pintado”. Se asume que la cadena no contiene caracteres `\n`.

  Cada caracter de la cadena ocupará 8x8 pixels; es decir que la imagen tendrá (8 * cantidad de caracteres) pixels de ancho y 8 pixels de alto.

  La información para pintar cada caracter está en `tipografia8x8`. Es un diccionario en el que la clave es un caracter ASCII y el valor son los 64 bits (agrupados en filas de 8) que representan al caracter ”pintado”.

  Cada uno de los pixels tomará el color de fondo o de texto según si el bit correspondiente es 0 o 1, respectivamente. Cada número de 8 bits representa una fila (el primer número es la fila de arriba), y el bit menos significativo es el pixel de la izquierda.

  Ejemplo, para el caracter >, en el diccionario tenemos:

  ```
  { 0x06, 0x0C, 0x18, 0x30, 0x18, 0x0C, 0x06, 0x00},
  ```

  El resultado en la imagen debe ser (`.` representa un pixel con color de fondo y `#` un pixel con color de texto):

  ```
  .##.....
  ..##....
  ...##...
  ....##..
  ...##...
  ..##....
  .##.....
  ........
  ```