UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Acceso al *framebuffer* en la plataforma Nintendo Switch

Apellidos, nombre	**Agustí i Melchor, Manuel** (magusti@disca.upv.es)
Departamento	**Departamento de Informática de Sistemas y Computadores (DISCA)**
Centro	Universitat Politècnica de València

1 Resumen de las ideas clave

Hablar del f*ramebuffer* es hablar de una posible solución a la necesidad de gestionar la información gráfica o visual que se muestra en un monitor de un sistema basado en computador. En la pantalla, las imágenes que se muestran, en digital, están formadas por representaciones de *F*x*C* píxeles. Donde *F* es el número de puntos de resolución en alto y *C* es el de ancho de la imagen que se muestra. En su concepción, este término hacía referencia al hardware que genera la señal de vídeo para una pantalla que contenía una memoria dedicada a mantener la información a mostrar. Con la evolución de las técnicas gráficas y el hardware que proporciona aceleración para estas, hemos ido llamando al hardware "subsistema de vídeo" o "tarjeta gráfica" y el término *framebuffer* se ha quedado como referencia al "lugar" donde se almacena la información a mostrar.

Así que es habitual entender el *framebuffer* como una única región de memoria que es actualizada por las aplicaciones y que contiene la siguiente imagen a mostrar en el monitor (*display*), por lo que el *framebuffer* es una zona contigua de memoria RAM de un tamaño de N elementos, que puede ser visto de forma:

- Lineal o secuencial. De longitud *N (*size *en la Figura 1a)* elementos, es decir del 0 al N-1, ambos incluidos.
- Plana o lógica. Se puede ver como el conjunto de valores que permiten indexar cada elemento (píxel) de que está compuesta la imagen, caracterizada esta por un ancho y un alto (*Figura 1b)*, *cuyo producto es el número de elementos*, *esto es*, *N = FxC*.

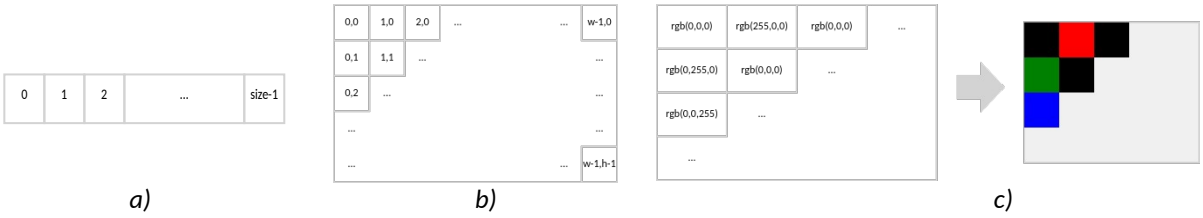


Figura 1: El Framebuffer es un área de memoria que guarda la información a mostrar en pantalla.

En el SDK no oficial [1] para desarrollar sobre la plataforma *Switch* podemos encontrar un ejemplo que accede al *framebuffer. Lo que nos permite comprobar* cómo se puede inicializar este modo gráfico de su pantalla, así como acceder y/o modificar el valor de un píxel directamente, esto es, que se puede trabajar en modo de acceso directo a la memoria de vídeo.

2 Objetivos

Una vez que el lector haya revisado este artículo con detenimiento y explore el código que se adjunta, dispondrá de una referencia para acceder directamente a la memoria de vídeo a través del *framebuffer*. En particular, será capaz de:

- Describir qué es el *framebuffer* en el contexto de la plataforma *Switch*.
- Describir cómo se inicializa o configura y cómo se accede, algorítmicamente, a bajo nivel (directamente), a los píxeles aplicado a la plataforma *Switch* a través de un ejemplo de código.
- Ampliar el ejemplo con la descripción de cómo convertir una imagen para cargarla directamente en el *framebuffer*.

3 Introducción

Como ya se ha indicado, hablamos de *framebuffer* como una memoria que permite una dualidad en el acceso a la misma. Por una parte, Como memoria RAM que es, su contenido es actualizado y consultado de forma aleatoria: esto es, puede ser accedido cada elemento de forma individual por parte de las aplicaciones que quieren llevar contenidos a la pantalla. Y, por otra parte, es recorrida con un determinado patrón que define el hardware que genera la señal de vídeo que llega al monitor. En este caso, *Figura 1c*, es la habitual señal en formato RGB con 8 bits por canal que describe la mayor parte de imágenes en formato de mapa de bits (también conocidas como *raster*). Así que con la información del *framebuffer* habrá una circuitería encargada de generar la señal de vídeo según el tipo de conexionado[1] que se utilice: analógica (circuito D/A) en componentes separadas o no, Figura 2a, o en digital, Figura 2b. La parte del hardware encargado de la conexión se le suele denominar *Display Controller* (DC) que solo lee del *framebuffer*, mientras que otros componentes de un computador, como el procesador central (CPU) o la tarjeta gráfica (Graphic Adapter po GA) acceden tanto para lectura como para escritura.

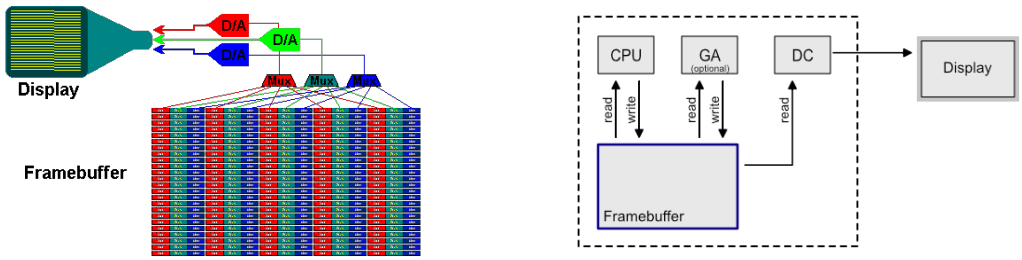
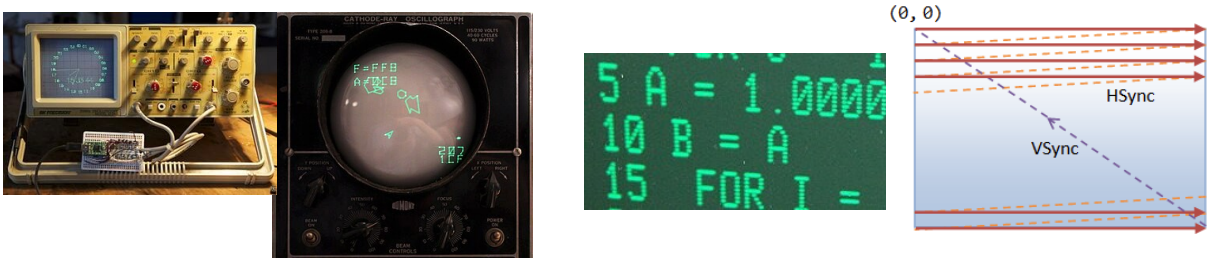


Figura 2: Arquitectura de un sistema basado en framebuffer. Imagen de "Frame Buffer Architecture of Raster Displays" <https://groups.csail.mit.edu/graphics/classes/6.837/F98/Lecture4/FramBuff.html>.

3.1 Un poco de historia

Las primeras técnicas de representación visual eran guiadas por operaciones de dibujo vectorial y la "pantalla" recibía nombres como *vector monitor* o *vector display*. En estas, la imagen se forman a partir de pintar líneas entre los puntos o vértices especificados por las primitivas gráficas, como se muestra en la Figura 3a y Figura 3b.

[1] Puede ampliar la información sobre conectores de vídeo en <https://es.wikipedia.org/wiki/Anexo:Conectores_de_video>.



a) b)

Figura 3: Ejemplos de dibujado: (a) vectorial todavía en uso en los osciloscopios analógicos (imagen de <https://en.wikipedia.org/wiki/Vector_monitor>) y (b) "raster scan" (de <https://en.wikipedia.org/wiki/Raster_scan>).

Posteriormente, llegaron las técnicas basadas en *raster scan*, que utilizan un patrón de generación de la imagen basado en recorrer está por líneas consecutivas (desde arriba hasta abajo) y pintando o no los puntos que se encuentra en ese recorrido horizontal. Actualmente, las necesidades gráficas de dibujado en 3D han llevado a utilizar APIs como OpenGL[2] (Figura 4a) o SDL que internamente gestionan esta memoria de vídeo, renderizando la escena tridimensional al *framebuffer* (bidimensional) mediante hardware o software. Estas y otras APIs de dibujado permiten técnicas más complejas como el "doble buffer" (Figura 4b) que permite seguir construyendo el próximo contenido a mostrar en pantalla, en el *backbuffer*, mientras se envía el actual por el DC a través del *frontbuffer*, Figura 4b.

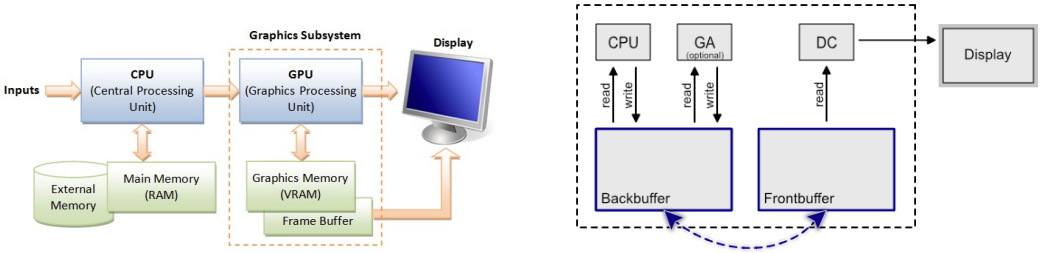


Figura 4: Arquitectura lógica de un sistema de representación gráfica que usa doble bufferframebuffer. Imagen de With Opengl The Basic Theory" <https://webframes.org/what-is-meant-by-frame-buffer-in-computer-graphics/>.

4 Desarrollo

Los aspectos básicos de acceso al hardware de la consola Switch están accesibles a través del interfaz que ofrece libnx [2]. Vamos a fijar nuestra atención en cómo se puede inicializar el modo gráfico de su pantalla, así como comprobar que se puede trabajar en modo de acceso directo a la memoria de vídeo. Para experimentar con el acceso a la memoria de vídeo empezaremos por ver el ejemplo de *Switch* dentro de *graphics/simplegfx*. Este ejemplo muestra el acceso a la memoria de vídeo (en modo *framebuffer*) asociada a la pantalla, con

[2] Se puede encontrar más información al respecto de OpenGL en su sitio web <https://www.opengl.org/> y al respecto de SDL en el suyo en <https://www.libsdl.org/>.

una resolución de 1280x720 píxeles va cambiando de nivel de gris, desde el negro hasta casi el blanco. La Figura 5 muestra algunas instantáneas de la secuencia.

Figura 5: Captura de cuatro instantes de la ejecución del ejemplo de Switch gráphics/simplegfx.

4.1 Uso del framebuffer en *Switch*

Vamos a describir el código que lo implementa y que se puede ver en el Listado 1. De momento, dejaremos a un lado las líneas que utilizan *DISPLAY_IMAGE*, como si no estuvieran ahí… Luego volvemos sobre ellas ¿de acuerdo?

A parte de las declaraciones de cabeceras habituales y de las constantes *FB_WIDTH* y *FB_HEIGHT* para declarar la resolución habitual de 1280x720[3], observe que no se inicializa el modo de vídeo en ningún momento; solo se crea una ventana (*nwindowGetDefault*, en la línea 14) y se crea un *framebuffer* asociado a ella (con *framebufferCreate*, en las líneas 16 y 17) con formato *PIXEL_FORMAT_RGBA_8888* que corresponde con la distribución de componentes de color y canal de transparencia (también denominado canal alfa) asociado a cada píxel, véase Figura 6, que asigna 8 bits a cada componente y otros tantos a la transparencia, por lo tanto 4*8 = 32 bits por pixel. El valor '2' en el último parámetro de la línea 17 especifica el número de *buffers* a crear. En la documentación de libnx [2] se indica: "1 for single-buffering, 2 for double-buffering or 3 for triple-buffering", así que utiliza la técnica del "*doble buffer*" (véase Figura 4b).

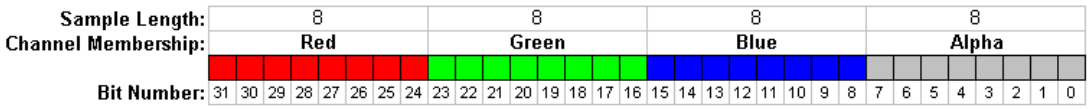


Figura 6: Formato RGBA. Imagen de "What is the difference between RGBA and ARGB?" <https://www.educative.io/answers/what-is-the-difference-between-rgba-and-argb>.

[3] La que se utiliza con la consola fuera de la base – el *dock* - donde sube hasta 1920x1080.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <switch.h>
#ifdef DISPLAY_IMAGE
#include "image_bin.h"//Your own raw RGB888 1280x720 image at "data/
image.bin" is required.
#endif
// Define the desired framebuffer resolution (here we set it to 720p).
#define FB_WIDTH  1280
#define FB_HEIGHT 720

int main(int argc, char* argv[]) { // Main program entrypoint
    NWindow* win = nwindowGetDefault();
    Framebuffer fb;
    framebufferCreate(&fb, win,
                      FB_WIDTH, FB_HEIGHT, PIXEL_FORMAT_RGBA_8888, 2);
    framebufferMakeLinear(&fb);
#ifdef DISPLAY_IMAGE
    u8* imageptr = (u8*)image_bin;
    const u32 image_width = 1280;
    const u32 image_height = 720;
#endif
    padConfigureInput(1, HidNpadStyleSet_NpadStandard);
    PadState pad;
    padInitializeDefault(&pad);
    u32 cnt = 0;
    while (appletMainLoop())     {      // Main loop
        padUpdate(&pad);
        u64 kDown = padGetButtonsDown(&pad);
        if (kDown & HidNpadButton_Plus) break;

        u32 stride;
        u32* framebuf = (u32*) framebufferBegin(&fb, &stride);
        if (cnt != 60) cnt ++; else cnt = 0;

        for (u32 y = 0; y < FB_HEIGHT; y ++)     {
            for (u32 x = 0; x < FB_WIDTH; x ++)             {
                u32 pos = y * stride / sizeof(u32) + x;
#ifdef DISPLAY_IMAGE
                if (y >= image_height || x >= image_width) continue;
                u32 imagepos = y * image_width + x;
                framebuf[pos] =
                   RGBA8_MAXALPHA(imageptr[imagepos*3+0]+cnt*4),
                       imageptr[imagepos*3+1], imageptr[imagepos*3+2]);
#else
                framebuf[pos] = 0x01010101 * cnt * 4;//... grey.
#endif
            }
        }
        framebufferEnd(&fb);
    }
    framebufferClose(&fb);
    return 0;
}
```

Listado 1: Código del ejemplo simplegfx.

Entre las líneas 24 y 26 se inicializa la gestión de eventos que se utiliza en las líneas 29 a la 31 para detectar la pulsación del botón '+' y terminar la aplicación. Si no es el caso, el bucle principal (que empieza en la línea 28) será el encargado averiguar la dirección de memoria del framebuffer creado (línea 34) y el *stride* (la longitud en bytes de las filas de píxeles en memoria, en este caso FB_WIDTH*4) del mismo. También actualizará el valor de ***cnt*** (linea 35), con lo que cada iteración incrementa esta variable en uno y, al llegar a 60, la resetea a cero. En la línea 37 empieza el doble bucle de recorrido de las filas y columnas del *framebuffer* para actualizar sus valores. Obteniendo la posición del elemento a modificar, línea 39 con la multiplicación del índice de fila por el número de elementos de una fila[4] más el valor de la columna. A partir de esa posición se modificarán los cuatro bytes consecutivos del framebuffer, que corresponden al RGBA de la posición en la fila **y**, columna **x**, del *framebuffer*.

La siguiente instrucción, en la línea 47, calcula un valor con un 1 en cada componente de color, lo que es un color prácticamente de 0 en RGB o lo que es lo mismo color negro. Este valor se multiplica por *cnt* y por 4, con lo que cada componente recibe un valor mayor, pero siempre igual en RGB, por lo que iremos viendo grises cada vez más claros. El último valor corresponde al máximo de ***cnt*** que es 60 (véase la línea 35) que multiplicado por 4 asigna 60*4=240 a cada componente de RGB, que corresponde a un gris muy claro. Obteniéndose los cambios vistos en la Figura 5.

Figura 7: Visualización de los cambios aplicados al código del ejemplo graphics/simplegfx.

Ejercicio 1: Compruebe que puede modificar el contenido que aparece en pantalla accediendo directamente a las posiciones de los píxeles y cambiado su contenido, p. ej., para obtener el resultado que se muestra en la Figura 7, sustituya la línea 47 del Listado 1 por

```
framebuf[pos] = RGBA8_MAXALPHA((u8)x, 0, 0);
```

La función ***RGBA8_MAXALPHA*** convierte los tres valores que recibe a RGBA8 (el formato de 32 bits por píxeles con que se ha inicializado el *framebuffer* de RGB y canal Alfa de transparencia) con el valor máximo de alfa (255), recuerde la Figura 6. Un detalle, la Figura 7 tiene la primera y la última columna a blanco, para remarcar dónde empieza y termina la figura,

[4] En la configuración del *framebuffer* del ejemplo siempre será el valor de stride de FB_WIDTH*4, por ser RGBA_8888. Al dividirlo por "sizeof(u32)" que es el tamaño en bytes de un unsigned int de 32 bits, se obtiene que sizeof(u32) = 4. Por lo tanto, la expresión es FB_WIDTH*4/4 = FB_WIDTH.

porque el emulador introduce unos bordes en negro. ¿Se le ocurre como añadir esas dos columnas diferentes? Compruébelo.

4.2 Cargar una imagen en el framebuffer

Ya habíamos avanzado que en el código hay unas referencias a *DISPLAY_IMAGE* que íbamos a examinar posteriormente. Ha llegado el momento. Examinemos el código y veremos que cuando este símbolo está definido se realizan acciones diferentes para dibujar el contenido en el *framebuffer*. Si está pensando que está cargando una imagen, está en lo cierto, estimado lector. Vamos a comprobarlo.

En las líneas 5 a la 8 del Listado 1 se ha podido observar que nos dice que si se ha definido el símbolo *DISPLAY_IMAGE*, hay que importar un fichero de cabecera (.h) de nombre image_bin.h y que, por el comentario, tiene que ver con una imagen propia en el subdirectorio *data* y con el nombre *image.bin*. Ahí lo tenemos, espera encontrar una imagen en formato binario (habitualmente denominadas *raw*, crudo o binario) que permita su lectura y carga en memoria sin más preámbulos[5]. Para ello en el proceso de compilación se lee la imagen en este formato sin cabecera y, eso sí, debe corresponder con la forma en que el código configura el *framebuffer* de la consola.

4.2.1 Conversión al formato del framebuffer

¿Cómo hacemos esa conversión de una imagen? ¿Qué formato espera la aplicación? Recuerde que sabemos que es RGBA_8888 y lo que significa, revise la Figura 6 si no lo recuerda. ¿Lo tiene? Y para obtener la imagen *raw*, se puede hace con *The Gimp* o con utilidades en línea de órdenes como *convert* (o *magick* en alguna plataforma) del conjunto de utilidades multiplataforma *ImageMagick*[6]. Empezaremos con esta última por los impacientes. Habrá de ejecutar las siguientes órdenes:

```
$ mkdir data
$ convert miImagen.png -depth 8 -alpha set  -resize 1280x720! rgba:miImagen.bin

$ cp -p miImagen.bin data/image.bin
```

Hemos utilizado una imagen PNG, pero puede utilizar una imagen JPG u otras que reconozca *convert*, simplemente cambiando el nombre de *miImagen.png* al de su fichero.

Por su lado, con *Gimp*, para generar este fichero binario y sin cabecera hemos de:

- Comprobar que tiene canal de transparencia (canal alfa), Figura 8.

[5] Si estuviera en un formato gráfico “de los habituales”, sería necesario código extra (propio o de alguna biblioteca de funciones) para leer el contenido, extraer los metadatos de la cabecera y, quizá, aplicar algún método de descompresión.

[6] En su equipo deberá comprobar su disponibilidad en su repositorio favorito de aplicaciones o en el sitio web de <https://imagemagick.org/>.

- Reescalar la imagen a la resolución configurada para *Switch*, 1280x720, Figura 9, y exportar.

- Cabe asegurarse al exportar que se escoge como formato “Datos de imagen en bruto (data,raw)”, Figura 10.

Y ya tenemos la imagen que hemos de guardar en *data/image.bin*.

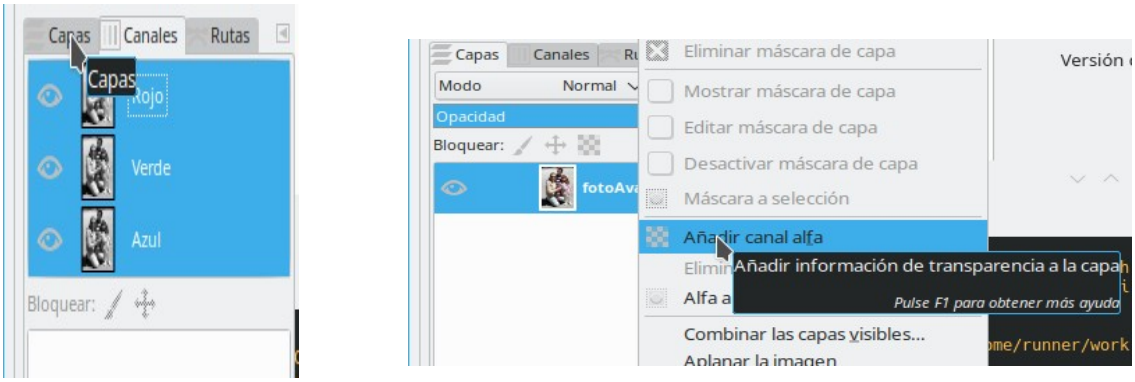


Figura 8: Comprobar que posee un canal alfa, si no hay que generarlo y escoger el color transparente.

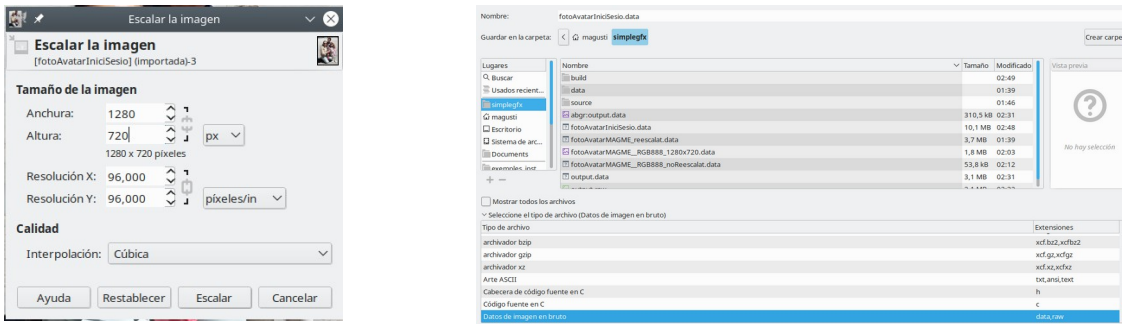


Figura 9: Reescalar y exportar desde Gimp.

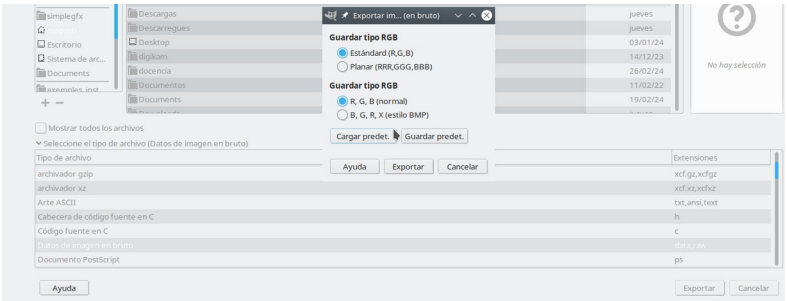


Figura 10: Exportar desde Gimp.

4.2.2 Ajustes necesarios en el código

Si ya tiene la imagen binaria copia en el subdirectorio data tendrá que modificar el fichero *Makefile*, donde dice

```
CFLAGS   +=       $(INCLUDE) -D__SWITCH__
 añadir
CFLAGS   +=      $(INCLUDE) -D__SWITCH__ -DDISPLAY_IMAGE
```

Con la orden *convert* o con *Gimp* habremos obtenido ya una versión "en crudo" (binaria) de la imagen escogida. Solo queda recompilar todo el proyecto:

```
$ make clean && make
```

```
1.  /* Generated by BIN2S - please don't edit directly */
2.  #pragma once
3.  #include <stddef.h>
4.  #include <stdint.h>
5.
6.  extern const uint8_t image_bin[];
7.  extern const uint8_t image_bin_end[];
8.  #if __cplusplus >= 201103L
9.  static constexpr size_t image_bin_size=3686400;
10. #else
11. static const size_t image_bin_size=3686400;
12. #endif
```

Listado 2: Ejemplo de contenido obtenido para image_bin.h.

Lo que generará (lo hace la utilidad *bin2s*, como se indica en el comentario de la línea 1 en el Listado 2) el archivo *image_bin.h* en el directorio *build*, dependiendo del tamaño de la imagen, su contenido será diferente, pero se parecerá al del Listado 2, en el que se puede ver el nombre de la estructura de datos que guarda los datos de la imagen (línea 6) y el tamaño, en bytes, de la misma, línea 9 u 11. El resultado, Figura 11a, va parpadeando porque utiliza la misma idea que el código sin cargar la imagen. ¿Recuerda la variable ***cnt***? Pero esta no es la imagen que yo he convertido... Seguro que a usted, estimado lector, le ha sucedido algo parecido.

a) *b)* *c)*

Figura 11: Resultado obtenido al ejecutar simplegfx: (a) con el código original, (b) con el código modificado y (c) sin el efecto de parpadeo (ejercicio 2).

Lo que ha sucedido es que la inicialización ha configurado un *framebuffer* de tipo RGBA (con transparencia, recuerde el Listado 1), también tendrá que modificar una línea de *main.c*, la que aparece entre las líneas 42 a 45 Listado 1, que deberá ser reescrita como:

```
  framebuf[pos] = RGBA8_MAXALPHA(imageptr[imagepos*4+0]+(cnt*4), ima-
geptr[imagepos*4+1], imageptr[imagepos*4+2]);
```

Recompile ahora la aplicación y verá que, ahora, los mensajes de compilación mencionan a *image.bin* y que el resultado parpadeante en pantalla muestra algo como la Figura 11b. Observe que al compilar, en el directorio *build*, se ha creado el fichero *image_bin.h* que se espera en el código fuente de *main.c* cuando se define *DISPLAY_IMAGE*.

Ejercicio 2: Busque una imagen propia para sustituirla por la que se ha proporcionado y siga las instrucciones del para generar la versión binaria de la imagen que ha escogido. Para que la salida deje de tener ese efecto estroboscópico vamos a modificar la línea de *main.c*.

```
   framebuf[pos] = RGBA8_MAXALPHA(imageptr[imagepos*4+0]+(cnt*4),
imageptr[imagepos*4+1], imageptr[imagepos*4+2]);
```

por

```
 framebuf[pos] = RGBA8_MAXALPHA(imageptr[imagepos*4+0],
imageptr[imagepos*4+1], imageptr[imagepos*4+2]);
```

que dejará la imagen tal cual está en el fichero, como se puede ver en la Figura 11c.

5 Conclusión y cierre

A lo largo de este objeto de aprendizaje hemos visto cómo es el acceso directo a la memoria de vídeo mediante el *framebuffer* para la plataforma N. *Switch*. El lector ha podido experimentar con una pequeña aplicación cómo acceder y modificar píxel a píxel el contenido que se muestra en pantalla, así como cargar el contenido de una imagen propia utilizando esta técnica.

Ahora, con estas reflexiones es cuestión de proponerse un ejemplo propio. Aprovechando lo que se muestra aquí (y el código de los ejemplos [1]) se puede comenzar un visualizador de imágenes, guardar capturas de pantalla de la salida de una aplicación o entender cómo funciona la captura o exportación a un formato de vídeo del servicio “Game Recording”. ¿Te animas, estimado lector?

6 Bibliografía y referencias

[1] devkitPro . <https://devkitpro.org/>.

[2] “Nintendo Switch AArch64-only userland library.”. Disponible en la URL <https://switchbrew.github.io/libnx/ >.