Programación Gráfica 2D (II)
Todo lo que quisiste saber sobre las superficies y nunca te atreviste a preguntar.

Autor: Sergio Hidalgo
serhid@wired-weasel.com

OJO: Este tutorial es algo mas "denso" que el anterior. Así que recomiendo que lo leais con la documentación de SDL en la mano, y quizás algún que otro tutorial con código que podais ir mirando.

Documentación de SDL, para el que aún no lo conozca:
http://www.libsdl.org/cgi/docwiki.cgi/SDL_20API

Introducción

En el tutorial anterior comenté qué era una superficie, y puse como crear una cargando sus datos desde un archivo. En este la idea es comentar más en detalle qué son las superficies, cómo se manejan realmente, y todo lo que se puede hacer con ellas.

En SDL, un SDL_Surface no es más que una estructura definida así:

```
typedef struct SDL_Surface {
   Uint32 flags;                              /* Read-only */
   SDL_PixelFormat *format;                   /* Read-only */
   int w, h;                                  /* Read-only */
   Uint16 pitch;                              /* Read-only */
   void *pixels;                              /* Read-write */
   SDL_Rect clip_rect;                        /* Read-only */
   int refcount;                              /* Read-mostly */
 /* This structure also contains private fields not shown here */
} SDL_Surface;
```

"flags" es una serie de indicadores que definen ciertas propiedades de la superficie. La lista completa está en la documentación, pero comentaré más adelante los más importantes.

"format" se refiere al formato del pixel, lo comentaré más abajo.

"w", y "h" son el ancho y alto en pixels de la superficie.

"pitch" es el ancho en bytes de la "scanline" de la superficie en memoria, esto también lo comentaré más abajo.

"pixels" es el buffer en memoria con los datos de la propia imagen.

"clip-rect" es un rectángulo de clipping. Sirve para definir un área "bliteable". Lo comentaré en el próximo tutorial.

"refcount" es simplemente un contador de referencias a la superficie, y no lo vamos a usar.

Creación de una superficie

En el otro tutorial puse el código para crear una superficie a partir de un archivo:

SDL_LoadBMP ("archivo.bmp");

En realidad, cuando creemos una superficie, nos interesa tener más control sobre su formato y sus propiedades. Por eso, al crearlas, usaremos la siguiente función:

```
SDL_Surface *SDL_CreateRGBSurface(Uint32 flags, int width, int height,
                                  int bitsPerPixel, Uint32 Rmask,
                                  Uint32 Gmask, Uint32 Bmask, Uint32 Amask);
```

Esto crea una superficie vacía, del tamaño indicado por width y height, y con las propiedades indicadas en flags. Los parámetros de "bitsPerPixel", y las máscaras dependen del formato del pixel, y lo comentaré más adelante.

De momento, vamos a ver los flags:

SDL_SWSURFACE: Hace que la superficie se cree en memoria del sistema
SDL_HWSURFACE: Hace que la superficie se cree en memoria de video
SDL_SRCCOLORKEY: Activa el color key para los blits desde esta superficie (lo comentaré en el siguiente tutorial)
SDL_SRCALPHA: Activa el alpha blending para los blits desde esta superficie (lo comentaré en el siguiente tutorial)

De momento los flags que nos interesan son los que se refieren al tipo de memoria en que habitará la superficie.

Tipos de Memorias

Hasta ahora decía que las superficies están "en la memoria". Pero en realidad, las superficies pueden crearse en dos tipos de memorias distintas: la memoria del sistema o la memoria de video.

La memoria del Sistema (SWSURFACE):
Con esto nos referimos a la memoria RAM clásica del ordenador. Por defecto al crear una superficie, a no ser que se indique lo contrario, se creará en esta memoria.
El microprocesador tiene un acceso directo a esta memoria, puede leer y escribir en ella sin problema, así que si planeamos estar accediendo a los datos de la superficie para modificarla "a mano" (sin blits), lo mejor será que esté aquí guardada.

La memoria de Video (HWSURFACE):
Esta memoria es parte de la tarjeta gráfica. El microprocesador no tiene acceso directo aquí, así que no puede modificar (o leer) los pixels de una superficie creada en esta memoria. Sin embargo, las superficies que están en memoria de video se benefician de la aceleración por hardware de la propia tarjeta, lo que hace que los blits se ejecuten mucho más rápido.

Las dos memorias se comunican a través de un bus (el bus AGP por lo general). Eso quiere decir que si hacemos un blit entre dos superficies, una en cada memoria, los datos tendrán que atravesar este bus, lo que es bastante más lento y puede dar lugar a cuellos de botella.

NOTA: A la hora de inicializar la librería, podemos elegir en qué tipo de memoria crearemos el Buffer Primario (con SDL_SetVideoMode). Hay que tener en cuenta que si lo creamos en la memoria del sistema, los datos tendrán que atravesar el bus para llegar hasta la tarjeta gráfica y mostrarse en pantalla, aunque SDL se encarga de que esto sea transparente al programador.

Resumiendo, a la hora de elegir memorias, hay que tener en cuenta:
Memoria del Sistema = Rápido acceso a los pixels, menos rapidez en blits.
Memoria de Video = Mayor velocidad de blits, mal acceso a los pixels.
Mucho tráfico de datos por el bus AGP = Cuello de botella

Con esto ya está aclarado (o eso espero), el tema de los flags a la hora de crear la superficie. Recuerdo que los otros dos los veremos en el siguiente tutorial.

```
SDL_Surface *SDL_CreateRGBSurface(Uint32 flags, int width, int height,
                                  int bitsPerPixel,Uint32 Rmask, Uint32 Gmask,
                                  Uint32 Bmask, Uint32 Amask);
```

Pero todavía queda el tema del formato de pixel y las máscaras

Formatos de Pixels

Cuando hablamos del formato del pixel, nos referimos a la manera en que un pixel guarda la información del color.

En primer lugar está la “profundidad de color”, o los “bits por pixel” (bpp). Esto es simplemente cuantos bits usamos para representar un pixel, y puede ser 8, 16, 24, o 32. A mas bits, mas rango de colores.

Con 8 bits solo tenemos 256 colores. Normalmente en este modo se suele usar una paleta, pero no quiero entrar en eso, porque no lo vamos a usar.

Con 16 bits o superior se considera representación “TrueColor”. A partir de aquí, lo que se guarda en el pixel son las componentes primarias del color: rojo, verde, y azul, o RGB(Red, Green, Blue).

Algunos formatos añaden también otro componente “alpha” que guarda la opacidad, muy útil para hacer efectos de transparencia. A esos formatos se les suele llamar RGBA.

Así que lo que diferencia a unos formatos de otros, a parte de la profundidad, es si tienen o no componente “alpha”, y como reparten los bits entre cada uno de los componentes.

Imaginad un formato para una profundidad de 32 bits. Teniendo 4 componentes (RGBA), podemos usar 8 bits para cada uno, entonces el pixel sería así:

RRRRRRRR GGGGGGGG BBBBBBBB AAAAAAAA

Y las máscaras:

11111111 00000000 00000000 00000000 = Rmask
00000000 11111111 00000000 00000000 = Gmask

00000000 00000000 11111111 00000000 = Bmask
00000000 00000000 00000000 11111111 = Amask

En hexadecimal:

Rmask = 0xff000000;
Gmask = x00ff0000;
Bmask = x0000ff00;
Amask = x000000ff

NOTA: Dependiendo del ordenador, hay memorias donde los bits mas significativos se colocan en las posiciones mas altas (es decir, al reves). Este tipo de memoria es la "Little Endian", y para saber en que modo estamos trabajando hay que consultar a SDL_BYTEORDER

Asi que dependiendo de ese indicador usaremos estas mascaras, o unas donde el rojo está en los bits menos significativos, el alpha en los más, etc...

```
#if SDL_BYTEORDER == SDL_BIG_ENDIAN
   rmask = 0xff000000;
   gmask = 0x00ff0000;
   bmask = 0x0000ff00;
   amask = 0x000000ff;
#else
   rmask = 0x000000ff;
   gmask = 0x0000ff00;
   bmask = 0x00ff0000;
   amask = 0xff000000;
#endif
```

En nuestro caso, probablemente usemos formatos de 16 bits. Los más comunes en este caso son: R5G6B5 (deja 1 bit mas para el verde), o R4G4B4A4 (con canal Alpha).

Toda esta información es la que se guarda en la estructura SDL_PixelFormat, que es un dato más de SDL_Surface. Esto quiere decir que cada superficie puede tener un formato de pixel distinto. Al hacer un blit el sistema automáticamente realiza la "traducción" entre formatos. Pero esa traducción tiene un coste en tiempo, claro, asi que lo mejor es evitarla si es posible.

Bueno, explicado el tema de las máscaras, ya sabemos como crear a pelo una superficie con las propiedades y formato que nos de la gana. Pero es un rollo. Hacer todo el tema de las máscaras para cada formato que podamos usar es un auténtico coñazo.

Pensemos un momento, lo que nos interesa es que todas las superficie compartan el mismo formato, ¿y qué formato es ese? Pues el que tenga el Buffer Primario. Cuando inicializamos la librería, se asigna un formato al buffer primario. Ese el formato que tenemos que recuperar y usar para cuando creemos nuestras propias superficies.

Para obtener ese formato, usaremos una función llamada SDL_GetVideoInfo (). Esto devuelve una estructura SDL_VideoInfo, con un campo llamado vfmt que es precisamente el formato de pixel del buffer primario. Y ese formato es una estructura que contiene las máscaras que tenemos que usar. ¿Suena raro? XD

Si mirais el código no es tan complicado:

```
const SDL_VideoInfo *vi;
vi = SDL_GetVideoInfo ();

superficie = SDL_CreateRGBSurface (SDL_HWSURFACE, w, h,
            vi->vfmt->BitsPerPixel,
            vi->vfmt->Rmask, vi->vfmt->Gmask,
            vi->vfmt->Bmask, vi->vfmt->Amask);
```

Las dos primeras lineas las ejecutariamos al principio del programa, y después guardaríamos el formato a lo largo de toda la ejecución, para no tener que llamar a GetVideoInfo cada dos por tres. Con todo esto ya hemos creado una superficie con las propiedades que queríamos, pero estará vacía. Ahora nos falta llenarla con los datos de un archivo, para eso lo que haremos será cargar el archivo en una superficie temporal, y copiar toda la imagen de ahí a "nuestra" superficie:

```
//Cargamos el archivo en la superficie auxiliar
temp = SDL_LoadBMP ("archivo.bmp");

//Creamos nuestra superficie
superficie = SDL_CreateRGBSurface (SDL_HWSURFACE, temp->w, temp->h,
            vi->vfmt->BitsPerPixel,
            vi->vfmt->Rmask, vi->vfmt->Gmask,
            vi->vfmt->Bmask, vi->vfmt->Amask);

//Copiamos los datos
SDL_BlitSurface (temp, NULL, superficie, NULL);

//Borramos la superficie auxiliar
SDL_FreeSurface (temp);
```

Y esto es todo. Tened en cuenta que en los valores de ancho (width), y alto (height), al crear la superficie estoy usando el ancho y alto de la superficie temporal, que son los del propio archivo.

Destrucción de una superficie

Igual que se crean, las superficies se destruyen cuando ya no nos hacen falta, liberando la memoria, para eso usamos SDL_FreeSurface ( ) como en el código anterior.

Rellenar una superficie de un color

Imaginad que queremos llenar una superficie, completamente, de color rojo turquesa. Para ello SDL nos da una función especial:

int SDL_FillRect(SDL_Surface *dst, SDL_Rect *dstrect, Uint32 color);

dst es la superficie que vamos a rellenar.
dstrect es el rectángulo dentro de la superficie que colorearemos (o NULL para colorearla entera)

Y por último, color es el color que usaremos... vale... y como hacemos que ese "int" signifique "rojo turquesa"?

Suponiendo que rojo turquesa en RGB sea por ejemplo (255, 120, 120), el valor que le pasamos a FillRect tiene que ser un entero que en el formato de pixel de la superficie se corresponda con el valor RGB que queremos.

Aquí podríamos entrar de nuevo con las máscaras y demás, pero por suerte, tenemos una función que se encarga de eso por nosotros:

Uint32 SDL_MapRGB(SDL_PixelFormat *fmt, Uint8 r, Uint8 g, Uint8 b);

Esta funcón toma un formato de pixel, y los valores RGB (de 0 a 255) del color que queramos. Lo que devuelve es un entero con el color representado en el formato indicado. Es decir, justo lo que necesitamos:

SDL_FillRect (superficie, NULL, SDL_MapRGB(superficie->format, 255, 120, 120));

OJO: Como formato de pixel uso el de la superficie, no el que sacamos de "VideoInfo" que había comentado antes. Esto es porque aunque en nuestro caso serán iguales, no tiene por qué ser siempre así.

Y para el que le interese añadir el canal alpha, también hay una función MapRGBA que hace lo mismo con un componente más.

Acceso directo al buffer de una superficie

Como en el otro tutorial, me dejo lo mejor para el final. Antes habíamos visto que el campo "pixels" en una SDL_Surface es el buffer propiamente dicho. Asi que si tenemos ahí el buffer con toda la información, ¿por qué no acceder directamente y modificar o leer los pixels a mano?

Bueno, de eso va este apartado.

Antes de poder acceder a los pixels, necesitamos "bloquear" la superficie. Bloquear no es nada más que poner una señal para asegurarnos de que ningún otro proceso va a acceder a ella mientras la estamos manipulando. Esto se hace mediante:

SDL_LockSurface(superficie)

Y por supuesto, al terminar, tendremos que desbloquearla con:

SDL_UnlockSurface(superficie);

NOTA: Si la superficie que bloqueamos está en memoria de video, el procesador no puede acceder directamente a ella, así que lo que SDL hace es copiar la superficie completa a la memoria del sistema. Al hacer Unlock, la vuelve a copiar a la memoria de video. Por eso esta memoria no es la ideal si tenemos que estar continuamente accediendo a las superficies a nivel de pixel.

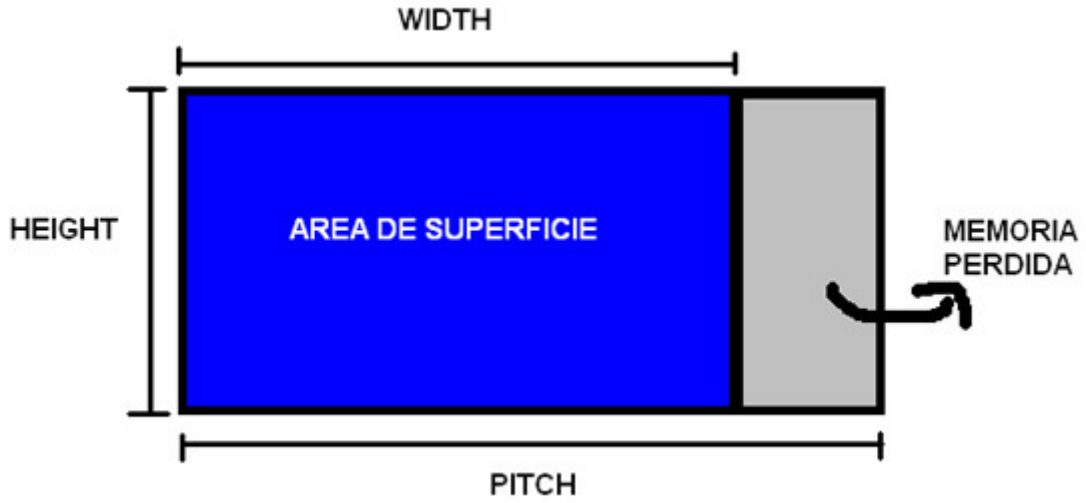
Una vez está bloqueada, ya podemos acceder al buffer. "pixels" es sencillamente un array con todas las "filas" de pixeles, puestas una a continuación de la otra. Es como si cojemos una matriz y la ponemos en forma de secuencia.

Asi que para leer el elemento x=4, y = 3, lo que hacemos es:

Array [3 * AnchoDeLaMatriz + (4 * Tamaño del elemento en bytes)]

Lo de multiplicar el 4 por el tamaño del elemento es porque nuestro buffer va en bytes, pero si usamos un formato de 16 bits, entonces cada pixel ocuparía 2 bytes, y tendríamos que multiplicar por 2.

Por otra parte, el "AnchoDeLaMatriz", no es lo mismo que el ancho de la superficie que tenemos. El tamaño de cada "linea" de pixels (o scanline en inglés) no se corresponde con la anchura de la superficie en pixels. Y aquí es donde entra otro de los campos de SDL_Surface que comenté al principio: **el pitch**



El pitch representa el ancho "real" de la superficie en memoria (viene dado en bytes), mientras que el ancho que manejábamos hasta ahora (w) representa la parte "visible" de la superficie.

La razón de tener esa zona de memoria perdida es por razones de alineamiento. De esta manera se consigue que cada "linea" esté alineada en memoria, y que el acceso sea más rápido.

Asi que resumiendo, para acceder a un pixel (x, y), haremos:

Array [(y * pitch) + (x * bytesPorPixel)]

Y ya en código, suponiendo que queramos escribir "color" sobre el pixel (x,y), y que estemos usando un formato de 16 bpp:

```
Uint16 *bufp;

//Avanzamos en bloques de 16 bits en vez de 1 byte
bufp = (Uint16 *)superficie->pixels + y*superficie->pitch/2 + x;
*bufp = color;
```

Y desde luego, "color" tendría que ser un valor generado a través de SDL_MapRGB y que se corresponda con el formato que estemos usando en la superficie.

Bueno, y aquí termina por hoy. Me dejo todo el tema del Clipping y ColorKeys para el siguiente tutorial, que irá sobre los blits.

Un saludo.