Capítulo 1

LOS PRIMEROS PASOS CON LA LIBRERÍA SDL

Introducción

Básicamente el curso está orientado a aprender a utilizar la librería SDL, para crear alguna demo o videojuego. Existen muchas librerías "allá afuera", muchas de ellas son comerciales y otras son gratuitas. De estas últimas existen bastantes, entre ellas tenemos a la conocida librería Allegro, la cual es bastante buena para comenzar en esto de la programación gráfica. También existen bastantes wrapper para DirectX. A mi parecer SDL es más profesional que Allegro, pero esto no quiere decir que con Allegro no se puedan hacer juegos buenos, ya que se han hecho bastantes, de muy buena calidad.

SDL significa **Simple DirectMedia Layer,** y fue creada por Sam Lantinga, Jefe de Programación de Loki Entertaiment Software, para portar juegos a Linux. Uno de los juegos portados es el Unreal Tournament, entre otros.

La SDL es una API multiplataforma, es decir, todas las aplicaciones que hagamos con esta librería pueden ser compiladas, muchas veces sin cambiar ninguna línea de código, en varias plataformas como: Linux, Windows, Macintosh, Beos, etc.

Requerimientos

Los requerimientos mínimos para seguir el curso son los siguientes:

- Saber programar en lenguaje C y C++. Por qué en C++ se preguntarán algunos, porque cuando sea posible encapsularemos algunas cosas en clases, aunque si solo saben C, igual pueden seguir el curso al principio, pero es recomendable que aprendan C++.
- Tener nociones básicas del uso de la STL (Standard Template Library), ya que nos servirá para utilizar listas enlazadas y arreglos de cualquier tamaño, entre otras cosas. Si no saben ocuparla, traten de buscar en la red algún tutorial sobre esto.
- Tener el compilador Mingw para windows o en su defecto el entorno de desarrollo integrado Dev-C++ que ya incluye este compilador, este último será el que usaremos.
- Ojalá tener algunos conocimientos básicos de programación gráfica. Si alguna vez programaron en el conocido modo 13h (320x200 a 256 colores) se les hará mucho más fácil. Y aún mejor si han utilizado la librería allegro u otra librería gráfica.

- Y por último, lo más importante, las librerías SDL, es decir los típicos archivos de cabecera (*.h,) y las librerías (*.lib).

Todas las herramientas necesarias para comenzar pueden ser descargadas de mi sitio web.

¿Por qué usar la SDL?

Muchos pueden pensar por qué usar la SDL para programar en Windows, si existen las librerías DirectX. Bueno la respuesta a esto es muy sencilla, primero SDL es más intuitiva y más fácil de usar que DirectX, segundo SDL es portable y DirectX no. Otros dirán está bien, pero con SDL no puedo hacer juegos 3D. Esto en cierta forma es verdad, pero no es tan así. Existe otra librería muy conocida, llamada OpenGL, la que está orientada a crear aplicaciones y videojuegos en 3 dimensiones. La ventaja de SDL es que puede usarse conjuntamente con OpenGL, pero eso ya es otro curso.

Preparando el compilador y la librería SDL

Lo primero que debemos hacer es instalar Dev-C++. No utilizaremos la IDE, ya que haremos todo mediante línea de comandos, aunque si quieren, pueden usarla. Una vez este instalado el Dev-C++, instalaremos las librerías SDL.

Una vez que hayan descomprimido el archivo de la librería SDL, deben copiar los directorios include y lib a los correspondientes en la carpeta del Dev-C++, por ejemplo en “c:\Dev-C++”. Una vez hecho esto, deben copiar el archivo llamado “sdl.dll” a “c:\windows\system” (si tienen win95 o win98) o a “c:\windows\system32” (si tienen windows NT, 2000, XP), todo esto suponiendo que el windows esta instalado en “c:\windows”.

Si alguien tiene problemas con la instalación del compilador o las librerías, no dude en preguntar en la lista de discusión.

Una vez que tengamos todo esto instalado, estamos listo para comenzar a programar.

Inicializando la librería

Todos los programas que hagamos comenzarán incluyendo el archivo **SDL/SDL.h** así:

#include <SDL/SDL.h>

Lo colocamos en mayúsculas, simplemente para que sea compatible con Linux, ya que este último es “case sensitive”.

Para inicializar la SDL debemos llamar a la función **SDL_Init(),** el prototipo de esta función es el siguiente:

int SDL_Init(Uint32 flags);

Esta función retorna -1 si algo falla y cero si tuvo éxito.

El parámetro que toma es para especificar que partes de SDL inicializar. Los más utilizados son los siguientes:

SDL_INIT_TIMER	Inicializa el subsistema timer
SDL_INIT_AUDIO	Inicializa el subsistema de audio
SDL_INIT_VIDEO	Inicializa el subsistema de video
SDL_INIT_CDROM	Inicializa el subsistema de cd-rom
SDL_INIT_JOYSTICK	Inicializa el subsistema de joystick
SDL_INIT_EVERYTHING	Inicializa todo lo anterior

Estos flags no necesariamente se pasan de a uno a la función, pueden ser pasados muchos de una vez, separándolos con el operador or. Por ejemplo para inicializar el subsistema de video y audio haríamos lo siguiente:

```
if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) < 0)
{
  fprintf(stderr, "Error al inicializar SDL: %s\n", SDL_GetError());
  exit(1);
}
```

Además deberíamos incluir el archivo **“stdlib.h”,** pues allí se encuentra el prototipo de la función exit();

Aquí apareció otra función la **SDL_GetError()**, lo que hace esta función es retornar un puntero a char, es decir un “string” que indica el tipo de error ocurrido. Una cosa importante, es que la función printf no imprime el mensaje en la pantalla, sino que lo escribe en un archivo llamado **stdout.txt**. En este caso utilizamos la función fprintf para que escriba en el archivo **stderr.txt**, el cual es usado para almacenar los errores ocurridos.

El prototipo de esta función es el siguiente:

char *SDL_GetError(void);

Otra función que siempre usaremos es la **SDL_Quit()**. Lo que hace esta función es cerrar todos los subsistemas creados inicialmente con SDL_Init() y libera los recursos ocupados por ellos. Su prototipo es:

void SDL_Quit(void);

Para no llamar explícitamente a la función SDL_Quit() antes de salir del programa, puede ser conveniente hacer lo siguiente, después de la función SDL_Init():

atexit(SDL_Quit);

Con esto al salir del programa automáticamente será llamada la función SDL_Quit().

Inicializando subsistemas

Anteriormente vimos como inicializar subsistemas usando la función SDL_Init(), pero existe otra forma, y esta es utilizar la función **SDL_InitSubSystem()** para inicializar un sistema en particular. Para cerrar este subsistema usamos la función **SDL_QuitSubSystem()**.

Los prototipos de estas funciones son:

int SDL_InitSubSystem(Uint32 flags);
void SDL_QuitSubSystem(Uint32 flags);

La función SDL_InitSubSystem() devuelve –1 si hubo problemas y 0 si tuvo éxito.

Por ejemplo, para inicializar el subsistema de Joystick haríamos lo siguiente:

```
if(SDL_InitSubSystem(SDL_INIT_JOYSTICK) < 0)
{
  fprintf(stderr, “No se encontró un joystick en el sistema: %s\n”, SDL_GetError());
  exit(1);
}
```

Y para cerrar este subsistema:

SDL_QuitSubSystem(SDL_INIT_JOYSTICK);

Pero existe otra cosa que puede suceder. Supongamos que queremos inicializar el sonido, pero si existe un error al inicializarlo debemos seguir en el programa, simplemente sin sonido. Una solución a esto es utilizar la función **SDL_WasInit(),** su prototipo es:

Uint32 SDL_WasInit(Uint32 flags);

Esta función nos permite saber que subsistemas han sido inicializados.

Por ejemplo para saber si el sonido fue inicializado, hacemos los siguiente:

```
Uint32 flags
bool sound;

flags=SDL_WasInit(SDL_INIT_EVERYTHING);
if(flags & SDL_INIT_SOUND) sound=true;
else sound=false;
```

Lo primero que hacemos es llamar a la función para que nos retorne una mascara de bits de todos los subsistemas inicializados. Luego vemos si está activado el bit del sonido, si es así hacemos true una variable booleana llamada sound, de lo contrario será false. De esta manera si en alguna sección de nuestro código queremos reproducir sonido, verificamos primero si sound es true.

Inicializar un modo de video

Antes de inicializar un modo de video en concreto veremos que son las superficies, un concepto importantísimo en la SDL.

Una superficie básicamente es una área de memoria, ya sea memoria del sistema (RAM) o memoria de video (de la tarjeta de video). En esta área de memoria podemos almacenar imágenes, dibujar sobre ella, etc.

En el caso de la SDL una superficie es una simple estructura, llamada **SDL_Surface**, definida de la siguiente forma:

```
typedef struct SDL_Surface {
    Uint32 flags;
    SDL_PixelFormat *format;
    int w, h;
    Uint16 pitch;
    void *pixels;
    SDL_Rect clip_rect;
    int refcount;
} SDL_Surface;
```

Ahora veremos que significa cada campo de esta estructura:

flags: son unos flags que indican el tipo de superficie, más adelante veremos estos tipos.
format: un puntero a una estructura que indica el formato de cada pixel, ya lo veremos.
w, h: ancho y alto de la superficie.
pitch: corresponde a la longitud de una línea de escaneo (scanline) de la superficie, medido en bytes.
pixels: puntero al comienzo de los datos (pixeles) de la superficie.
clip_rect: estructura que indica el área (un simple rectángulo) de clipping de la superficie, también la veremos muy luego.
refcount: contador de referencia, usado cuando se libera la superficie.

Todos los campos anteriores son de solo lectura. En el único que se puede escribir es en el puntero pixels. Este será usado más adelante para colocar pixeles en una superficie.

Ahora veremos la definición de la estructura **SDL_PixelFormat**:

```
typedef struct{
  SDL_Palette *palette;
  Uint8  BitsPerPixel;
  Uint8  BytesPerPixel;
  Uint32 Rmask, Gmask, Bmask, Amask;
  Uint8  Rshift, Gshift, Bshift, Ashift;
  Uint8  Rloss, Gloss, Bloss, Aloss;
  Uint32 colorkey;
  Uint8  alpha;
} SDL_PixelFormat;
```

El significado de cada campo es el siguiente:

palette: es un puntero a la paleta. Es un estructura (ya la veremos). Vale NULL si los bits por pixel son superiores a 8. Esto se debe a que los modos de 16, 24 y 32 bits no utilizan paleta.
BitsPerPixel: es el número de bits usado para representar cada pixel en la superficie. Usualmente es 8, 16, 24 o 32 bits.
BytesPerPixel: Como el anterior pero ahora medido en bytes. Pueden ser 1, 2, 3 o 4 bytes.
Rmask, Gmask, Bmask, Amask: mascaras binarias usadas para obtener los valores de cada componente del color (rojo, verde, azul o alpha).
Rshift, Gshift, Bshift, Ashift: desplazamiento binario hacia la izquierda de cada componente de color en el valor del pixel.
Rloss, Gloss, Bloss, Aloss: pérdida de precisión de cada componente de color ($2^{[RGBA]loss}$).
colorkey: aquí se almacena el color que es transparente en la superficie. Es usado cuando no queremos que cierto color de la superficie se muestra. Esto lo utilizaremos más adelante cuando trabajemos con sprites.
alpha: valor alpha de la superficie.

Anteriormente vimos que aparecía una estructura denominada **SDL_Palette**, su definición es la siguiente:

```
typedef struct{
  int ncolors;
  SDL_Color *colors;
} SDL_Palette;
```

El significado de cada campo es el siguiente:

ncolors: número de colores usados en la paleta.
colors: puntero a una estructura (es un arreglo de estructuras SDL_Color) que almacena los colores que forman la paleta.

La paleta es solo usada en los modos de 8 bits por pixel, es decir los modos que funcionan a 256 colores.

Ahora veremos la definición de la estructura **SDL_Color**:

```
typedef struct{
  Uint8 r;
  Uint8 g;
  Uint8 b;
  Uint8 unused;
} SDL_Color;
```

Y el significado de cada campo:

r: intensidad de la componente roja.
g: intensidad de la componente verde.
b: intensidad de la componente azul.
unused: no utilizado.

Y por último nos está quedando ver una de las estructuras más simples, pero a la vez muy usada, esta es la **SDL_Rect**, la cual define un área rectangular, veamos su definición:

```
typedef struct{
  Sint16 x, y;
  Uint16 w, h;
} SDL_Rect;
```

Y como siempre, veremos que significa cada campo:

x, y: posición de la esquina superior izquierda del rectángulo.
w, h: ancho y alto del rectángulo.

Existen otras estructuras en la SDL, pero las que vimos son las más utilizadas.

Luego de ver tantas estructuras, podremos inicializar nuestro modo de video favorito, siempre y cuando esté soportado por nuestra tarjeta de video. La función que hace este trabajo es **SDL_SetVideoMode()**, y su prototipo es el siguiente:

SDL_Surface *SDL_SetVideoMode(int width, int height, int bpp, Uint32 flags);

Veamos como funciona. Esta función devuelve un puntero a una superficie, y es la más importante, ya que en esta superficie dibujaremos todo lo que queremos que aparezca en pantalla, lo único que debemos hacer después de dibujar en ella, es actualizar el área en donde dibujamos, ya veremos como se hace eso.

Los parámetros son cuatro, los dos primeros corresponden al ancho y alto (medido en pixeles) del modo se video que queremos iniciar. El tercero es el número de bits por pixel que queremos, puede ser 8, 16, 24 o 32 bits, siendo el más utilizado el de 16 bits. Y por último el cuarto parámetro indica el tipo de superficie que queremos crear. Si recuerdan, la estructura SDL_Surface tenía un campo llamado flags, bueno esta es la única forma que tenemos para modificarla, ya que es de solo lectura. Si hubo un error al inicializar el modo de video, la función devuelve NULL.

La función SDL_Quit() libera la memoria ocupada por esta superficie.

Ahora veremos una tabla con los flags disponibles. Al igual como ocurría con los flags SDL_IINIT_*, se pueden pasar varios de una vez utilizando el operador or.

SDL_SWSURFACE	Crea la superficie en la memoria del sistema (RAM).
SDL_HWSURFACE	Crea la superficie en la memoria de video (Tarjeta de video).
SDL_ASYNCBLIT	Habilita el uso del volcado asíncrono a la superficie de visualización. Esto provocará usualmente la ralentización del volcado en máquinas de una única CPU, pero puede aumentar la velocidad en sistemas SMP.
SDL_ANYFORMAT	Normalmente, si una superficie de video de la profundidad requerida (bpp) no está disponible, SDL emulará una con una superficie en sombras (shadow surface). Pasando SDL_ANYFORMAT se evita esto y causa que SDL use la superficie de vídeo, independiente de la profundidad.
SDL_HWPALETTE	Permite el acceso exclusivo a la paleta. Sin este flag no siempre podrás obtener colores utilizando las funciones SDL_SetColors() y SDL_SetPalette().
SDL_DOUBLEBUF	Habilita la utilización de búfer doble. Una llamada a SDL_Flip() intercambiará los búferes y actualizará la pantalla. Si la utilización de búfer doble no se encuentra habilitada, entonces SDL_Flip() simplemente realizará un SDL_UpdateRect() en toda la pantalla
SDL_FULLSCREEN	SDL intentará usar el modo a pantalla completa. Esto no siempre es posible.
SDL_OPENGL	Crea un contexto de renderizado OpenGL. Se deben haber establecido previamente los atributos OpenGL con SDL_GL_SetAttribute().
SDL_OPENGLBLIT	Como el anterior pero permite operaciones de volcado normales.
SDL_RESIZABLE	Crea una ventana de tamaño modificable. Si la ventana cambia de tamaño se generará el evento SDL_VIDEORESIZE y se puede llamar nuevamente a la función SDL_SetVideMode(), para cambiar al nuevo tamaño.
SDL_NOFRAME	Si es posible, SDL crea un ventana sin barra de título u otras decoraciones. El modo FullScreen activa automáticamente este flag.

De todos estos flags los más usados son el SDL_HWSURFACE o SDL_SWSURFACE con SDL_DOUBLEBUF. Y muchas veces también junto a SDL_FULLSCREEN, aunque este último no siempre funciona en algunas tarjetas de video.

Veamos ahora como inicializar un modo de video concreto. Por ejemplo veremos como inicializar el modo de video a 640x480 pixeles y 16 bits de color, utilizando la memoria de video y double buffer:

```
SDL_Surface *screen;

screen=SDL_SetVideoMode(640, 480, 16, SDL_HWSURFACE | SDL_DOUBLEBUF);
if(screen==NULL)
{
  fprintf(stderr, "Error al inicializar modo de video: %s", SDL_GetError());
  exit(1);
}
```

Lo primero que hacemos aquí es crear un puntero a una estructura SDL_Surface llamada screen. Luego llamamos a la función SDL_SetVideoMode() con los parámetros respectivos y le asignamos la superficie devuelta a screen. Luego si screen es NULL significa que hubo problemas al inicializar ese modo de video. Una de las causas por la que ocurre este error, es debido a que el modo no está soportado por la tarjeta de video.

Obteniendo información de la tarjeta de video

Podemos saber que características soporta nuestra tarjeta de video. Para hacer esto debemos llamar a la función **SDL_GetVideoInfo()**, su prototipo es:

```
SDL_VideoInfo *SDL_GetVideoInfo(void);
```

Vemos que devuelve un puntero a una estructura llamada **SDL_VideoInfo**, el cual es de solo lectura. Para poder conocer las características de nuestra tarjeta de video, debemos también conocer como esta formada esta estructura, veámosla:

```
typedef struct {
  Uint32 hw_available:1;
  Uint32 wm_available:1;
  Uint32 blit_hw:1;
  Uint32 blit_hw_CC:1;
  Uint32 blit_hw_A:1;
  Uint32 blit_sw:1;
  Uint32 blit_sw_CC:1;
  Uint32 blit_sw_A:1;
  Uint32 blit_fill;
  Uint32 video_mem;
  SDL_PixelFormat *vfmt;
} SDL_VideoInfo;
```

Y la explicación de cada campo de la estructura es la siguiente:

hw_available: indica si es posible crear superficies en la memoria de video.
wm_available: indica si hay disponible un manejador de ventanas. En WIN32 este valor es uno, pero en otras plataformas puede que no sea así. El manejador de ventanas es otro subsistema de SDL, que se encarga de configurar algunas cosas para una ventana, por ejemplo para un modo windowed o fullscreen.
blit_hw: especifica si está disponible la aceleración para hacer volcados entre superficies que se encuentran en la memoria de video.
blit_hw_CC: como blit_hw para superficies con partes transparentes.
blit_hw_a: como blit_hw para superficies alpha.
blit_sw: como blit_hw pero aquí se realizan volcados de superficies de memoria del sistema a la memoria de video.
blit_sw_CC: como blit_sw para superficies con partes transparentes.
blit_sw_A: como blit_sw para superficies alpha.
blit_fill: especifica si las operaciones de rellenado de color son aceleradas.
video_mem: cantidad total de memoria de video, medida en Kilobytes.
vfmt: es un puntero a la estructura SDL_PixelFormat, que contiene el formato de pixel de la tarjeta de video. Si se llama a SDL_GetVideoInfo() antes de SDL_SetVideoMode(), esta estructura contendrá el formato de pixel del mejor modo de video.

Por ejemplo si queremos conocer si existe aceleración para realizar volcados entre superficies en la memoria de video, hacemos lo siguiente:

```
const SDL_VideoInfo *info;

info=SDL_GetVideoInfo();
if(info->blit_hw)
  printf("Aceleracion de volcados de superficies disponible\n");
else
  printf("Aceleracion de volcados de superficies no disponible\n");
```

La variable info fue declarada como const (constante) ya que la función SDL_GetVideoInfo() devuelve un puntero de solo lectura.

Otra función interesante es la **SDL_VideoDriverName()**, la cual devuelve el nombre del driver de video, su prototipo es el siguiente:

```
char *SDL_VideoDriverName(char *namebuf, int maxlen);
```

Esta función toma dos parámetros, el primero tiene que ser un puntero a una cadena de caracteres, en donde se guardará el nombre del driver. El segundo corresponde a la máxima cantidad de caracteres para almacenar el nombre, incluido el carácter NULL. La función retorna NULL si el subsistema de video no ha sido inicializado son SDL_Init, de lo contrario es devuelto un puntero al nombre del driver.

Un ejemplo del uso de esta función es el siguiente, suponiendo que la librería ya ha sido inicializada:

```
char nombre_driver[81];

SDL_VideoDriverName(nombre_driver, 81);
printf("Driver tarjeta de video: %s\n", nombre_driver);
```

Otra función muy útil es **SDL_ListModes()** la cual retorna un puntero a un arreglo de los modos disponibles en el sistema, su prototipo es:

```
SDL_Rect **SDL_ListModes(SDL_PixelFormat *format, Uint32 flags);
```

Esta lista que devuelve se almacena en estructuras SDL_Rect, las que contienen los modos de video ordenados de mayor a menor. El primero parámetro es un puntero a una estructura SDL_PixelFormat. Este valor puede ser NULL o el valor devuelto por SDL_GetVideoInfo()->vfmt. El Segundo parámetro es una combinación de flags (por ejemplo, algunos de los vistos anteriormente, correspondientes al tipo de superficie).

Además la función puede retornar NULL si no hay modos disponibles para un formato en particular, o –1 si cualquier modo es válido para un tipo de formato dado.

En general lo que se hace es llamar a la función con los flags que queremos comprobar. Por ejemplo podemos pasarle SDL_FULLSCREEN, para ver que modos están soportados a pantalla completa.

Un ejemplo de esto puede ser el siguiente:

```
SDL_Rect **modos;

modos=SDL_ListModes(NULL, SDL_FULLSCREEN);
if(modos==(SDL_Rect **)0)
  printf("No existen modos disponibles\n");
else if(modos==(SDL_Rect **)-1)
  printf("Todos los modos disponibles\n");
else
{
  printf("Lista de modos disponibles:\n");
  for(int i=0; modos[i]; i++)
  printf("%dx%d\n", modos[i]->w, modos[i]->h);
}
```

Y por último existe otra función para verificar si un modo en particular es soportado, esta función es la **SDL_VideoModeOK()** y su prototipo es:

```
int SDL_VideoModeOK(int width, int height, int bpp, Uint32 flags);
```

Se le debe pasar las dimensiones del modo a consultar (ancho y alto), el número de bits por pixel y en flags las características del modo. Retorna 0 si el modo no está soportado, de lo contrario retorna el número de bits por pixel.

Es una buena práctica consultar si el modo está disponible, antes de inicializarlo con la función SDL_SetVideoMode().

Ahora veamos un ejemplo sencillo de esto:

```
SDL_Surface *screen;

if(!SDL_VideoModeOK(640, 480, 16, SDL_HWSURFACE))
{
  printf("El modo 640x480x16 no está disponible\n");
  exit(1);
}

screen=SDL_SetVideoMode(640, 480, 16, SDL_HWSURFACE);
```

¿Como colocar un pixel en la pantalla?

Colocar pixeles en la pantalla es una de la cosas más básicas que podemos hacer en cualquier programa gráfico. En SDL hay ciertas cosas a tener en cuenta antes de hacerlo. Primero cuando queremos colocar un pixel en una cierta posición de una superficie, debemos verificar si esta superficie debe ser bloqueada. Si esto es cierto bloqueamos la superficie en la que dibujaremos, y una vez que se haya dibujado, la superficie deberá ser desbloqueada.

Como vimos lo primero es ver si la superficie debe ser bloqueada, para esto utilizaremos la macro **SDL_MUSTLOCK(surface)** que devuelve 1 si debe ser bloqueada o cero en caso contrario. Se le debe pasar un puntero a una estructura SDL_Surface.

Si la superficie debe ser bloqueada utilizaremos la función **SDL_LockSurface(),** su prototipo es el siguiente:

int SDL_LockSurface(SDL_Surface *surface);

Simplemente se la pasa un puntero a la superficie que queremos bloquear. Retorna 0 si pudo ser bloqueada y –1 en caso contrario.

Una vez que hayamos dibujado en la superficie esta deberá ser desbloqueada, siempre y cuando sea necesario, para realizar esta tarea utilizaremos la función **SDL_UnlockSurface()**, su prototipo es:

void SDL_UnlockSurface(SDL_Surface *surface);

A esta función, al igual que la anterior, solo se le pasa un puntero a la superficie que debe ser desbloqueada.

Veamos un ejemplo de esto:

```
SDL_Surface *screen;

If(SDL_MUSTLOCK(screen))
  SDL_LockSurface(screen);

// Dibujamos algo

If(SDL_MUSTLOCK(screen))
  SDL_UnlockSurface(screen);
```

Bien, ahora que ya sabemos como bloquear y desbloquear una superficie, veremos como colocar un pixel.

Colocar un pixel en una superficie es muy sencillo, pero depende del número de bits por pixel de la superficie. Para efectos prácticos veremos primero un ejemplo de como colocar un pixel en la pantalla (screen), la cual será una superficie de 8 bits, esto es similar a como se hacía en el modo 13h (320x200x256).

Supongamos que tenemos dos variables para la posición (x e y), otra para el color del pixel y por último un puntero a la memoria en donde colocaremos nuestro pixel. Veamos:

```
int x, y;
Uint32 color;
Uint8 *buffer;
SDL_Surface *screen;

c=10; y=10;
color=SDL_MapRGB(screen->format, 255, 0, 0);

buffer=(Uint8 *) screen->pixels+y*screen->pitch+x;
*buffer=color;
```

Lo que hacemos aquí simplemente es colocar un pixel de color rojo en la posición (10, 10), suponiendo que screen ya ha sido inicializado. Es bueno recordar que hay que bloquear y desbloquear la superficie antes de realizar esta tarea, cuando sea necesario. Aquí no lo hicimos simplemente para no hacer más largo el ejemplo.

Pero, ¿cómo obtenemos el color?, bueno esto lo hacemos con la función **SDL_MapRGB()**, su prototipo es el siguiente:

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

Como vemos, primero se necesita un puntero al formato del pixel, pero esto lo sacamos de la variable format, que lo contiene la estructura SDL_Surface. Luego se necesita la intensidad de cada componente (rojo, verde, azul).

La función retorna un entero que contiene el color pedido. Si el BPP es de 16 bits el valor retornado será de tipo Uint16 y si es de 8 bits será de tipo Uint8.

Ahora crearemos una función llamado **putpixel()**, la que estará encargada de colocar un pixel en cualquier posición de una superficie determinada. El prototipo de está función es:

```
void putpixel(SDL_Surface *screen, int x, int y, SDL_Color color);
```

Esta función toma como parámetros un puntero a la superficie, la posición del pixel y el color de este, el cual será una estructura de tipo SDL_Color.

La ventaja de hacer esta función, es que podemos colocar un pixel en cualquier superficie, y esta podrá tener cualquier número de bytes por pixel. Para saber el número de BPP consultaremos la variable BytesPerPixel de la estructura SDL_PixelFormat.

Veamos ahora la implementación de esta función:

```
void putpixel(SDL_Surface *screen, int x, int y, SDL_Color color)
{
  // Convertimos color
  Uint32 col=SDL_MapRGB(screen->format, color.r, color.g, color.b);

  // Determinamos posición de inicio
  char *buffer=(char*) screen->pixels;

  // Calculamos offset para y
  buffer+=screen->pitch*y;

  // Calculamos offset para x
  buffer+=screen->format->BytesPerPixel*x;

  // Copiamos el pixel
  memcpy(buffer, &col, screen->format->BytesPerPixel);
}
```

Obteniendo un píxel desde la pantalla

Ahora haremos el proceso inverso a putpixel(). Conociendo la posición de un píxel, queremos conocer su color.

El prototipo de nuestra función será el siguiente:

SDL_Color getpixel(SDL_Surface *screen, int x, int y);

Esta función devolverá una estructura de tipo SDL_Color. El primer parámetro es un puntero a la superficie, y los otros dos la posición del pixel.

Como devolvemos una estructura SDL_Color debemos descomponer el color en sus componentes rojo, verde y azul . Para esto usaremos la función **SDL_GetRGB()**, su prototipo es:

void SDL_GetRGB(Uint32 pixel, SDL_PixelFormat *fmt, Uint8 *r, Uint8 *g, Uint8 *b);

Como vemos el primer parámetro es el color o píxel que queremos descomponer. El segundo parámetro es el formato del píxel, y los tres restantes serán las componentes del color que buscamos.

Veamos ahora la implementación de nuestra función getpixel():

```
SDL_Color getpixel(SDL_Surface *screen, int x, int y)
{
  SDL_Color color;
  Uint32 col;

  // Determinamos posición de inicio
  char *buffer=(char *) screen->pixels;

  // Calculamos offset para y
  buffer+=screen->pitch*y;

  // Calculamos offset para x
  buffer+=screen->format->BytesPerPixel*x;

  // Obtenemos el pixel
  memcpy(&col, buffer, screen->format->BytesPerPixel);

  // Descomponemos el color
  SDL_GetRGB(col, screen->format, &color.r, &color.g, &color.b);

  // Devolvemos el color
  return color;
}
```

Actualizando la pantalla

Muchas veces necesitaremos actualizar la pantalla, después de haber dibujado algo en ella. Existen dos funciones para realizar esta tarea, una de ellas es **SDL_UpdateRect()**, la cual actualiza un área de una superficie, su prototipo es:

void SDL_UpdateRect(SDL_Surface *screen, Sint32 x, Sint32 y, Sint32 w, Sint32 h);

El primer parámetro es un puntero a la superficie que queremos actualizar. Los siguientes cuatro parámetros forman el área de la superficie que será actualizada.

Hay que tener en cuenta que el área que será actualizada debe estar dentro de los límites de la superficie, ya que no se hace clipping. Si x, y, w y h son cero la superficie entera será actualizada. Por último esta función no debe ser llamada si la superficie está bloqueada.

También existe otra función similar a la anterior, esta es **SDL_UpdateRects()**, su prototipo es:

void SDL_UpdateRects(SDL_Surface *screen, int numrects, SDL_Rect *rects);

La única diferencia es que permite actualizar varias áreas de la superficie al mismo tiempo. El primer parámetro es un puntero a la superficie, el segundo es el número de rectángulos que se quieren actualizar y por último el tercer parámetro es un arreglo de rectángulos. Es conveniente llamar a esta función una vez por ciclo, para que no se produzca un proceso de sobrecarga.

Por último tenemos la función **SDL_Flip()**, la cual es usada cuando inicializamos un modo gráfico que soporta double buffer, es decir la inicializamos con el flag SDL_DOUBLEBUF. Su prototipo es el siguiente:

int SDL_Flip(SDL_Surface *screen);

Lo que hace esta función es intercambiar los dos buffer, y mostrar uno en la pantalla, dejando el otro para dibujar sobre el. Además esta función espera el retrazo vertical antes de intercambiarlos. Si el double buffer no está soportado por hardware, llamar a esta función es equivalente a una llamada a **SDL_UpdateRect(screen, 0, 0, 0, 0)**. Esta función retorna cero si todo ha salido bien, y –1 si ha habido algún error.

Nuevos “tipos de datos” en SDL

Muchos pueden preguntarse que es Uint8 o Sint16, etc. Bueno, como ya saben SDL es una librería portable a otros sistemas, por tal razón algunos tipos de datos no tienen el mismo tamaño en todas las plataformas. Es por esto que en SDL existen algunos typedefs de las variables que pueden sufrir modificaciones en sus tamaños. Veamos a continuación un tabla de equivalencias de estos typedefs versus los tipos de datos comunes que todos ya deben conocer:

Uint8	unsigned char a 8 bit (1 byte)
Uint16	unsigned int a 16 bit (2 byte)
Uint32	unsigned int a 32 bit (4 byte)
Uint64	unsigned int a 64 bit (8 byte)
Sint8	signed char a 8 bit (1 byte)
Sint16	signed int a 16 bit (2 byte)
Sint32	signed int a 32 bit (4 byte)
Sint64	signed int a 64 bit (8 byte)

Resumen

Bueno hemos llegado al final del primer capítulo de este curso, salió bastante extenso, ya que vimos bastantes estructuras y funciones de SDL. En realidad quise hacerlo lo más completo posible para que este capítulo les sirva de referencia en el futuro cuando programen.

En este capítulo aprendimos que es SDL, quién lo hizo y por qué existe. Aprendimos también como inicializarla, inicializar modos gráficos, ver cuales de estos se encuentran disponibles, obtener información de la tarjeta de video, cómo colocar y obtener pixeles de la pantalla, y cómo actualizarla después se haber dibujado en ella. Y por último vimos los nuevos typedefs que define SDL.

En el próximo capítulo veremos cosas más interesantes como el manejo de ventanas, es decir, como colocar un icono, un título a una ventana, manejo de eventos, carga de imágenes y manejo del teclado. Haremos nuestras primeras animaciones.

A este capítulo lo acompañan dos ejemplos en SDL, uno de ellos muestra como obtener información de nuestra tarjeta de video y de algunos modos gráficos. El otro coloca pixeles al azar en la pantalla. En este último ejemplo hago uso del manejo de ventanas y de los eventos para poder mantenerme en un ciclo. No se preocupen si no entienden algo, ya que esto, como dije, lo veremos en el próximo capítulo.