

Capítulo 2

CONOCIENDO MÁS A FONDO LA SDL

Manejo de ventanas

Ahora veremos como decorar nuestra ventana para que se vea de una forma más profesional. Primero veremos un función para colocar un título en una ventana. Esta función es **SDL_WM_SetCaption()** y su prototipo es el siguiente:

```
void SDL_WM_SetCaption(const char *title, const char *icon);
```

Como vemos es una función muy sencilla de utilizar. Primero se le pasa un cadena de caracteres que será el título que aparece en la parte superior de la ventana (barra de título). El segundo parámetro corresponde al nombre del icono de la ventana, que frecuentemente se le pasará NULL.

Por ejemplo si quisiéramos asignarle a la barra de título “Mario Bros”, simplemente haríamos esto:

```
SDL_WM_SetCaption(“Mario Bros”, NULL);
```

Como se vio en la función anterior no le asignamos ningún icono a la ventana, ya que le pasamos NULL. La función que utilizaremos para esto será **SDL_WM_SetIcon()**, su prototipo es:

```
void SDL_WM_SetIcon(SDL_Surface *icon, Uint8 *mask);
```

El primer parámetro corresponde a un puntero a una superficie. El segundo parámetro es una mascara en formato MSB. Si el parámetro mask es NULL, la superficie completa del icono se usará como icono. Hay que tener en cuenta que esta función deberá ser llamada antes de la primera llamada a **SDL_SetVideoMode()**. En Win32 el tamaño del icono deberá ser de 32x32 pixeles.

Frecuentemente para asignarle un icono a nuestra ventana usaremos la función anterior junto a **SDL_LoadBMP()**, por ejemplo:

```
SDL_WM_SetIcon(SDL_LoadBMP(“icono.bmp”), NULL);
```

Donde “icono.bmp” es simplemente una imagen en formato bmp, que contiene al icono.

Otra función (muy poco utilizada) es **SDL_WM_GetCaption()**, la cual obtiene el nombre del título de la ventana y del icono. Su prototipo es:

```
void SDL_WM_GetCaption(char **title, char **icon);
```

Por ejemplo si quisiéramos conocer estos nombres se podría hacer algo así:

```
char title[256], icono[256];
```

```
SDL_WM_GetCaption(&title, &icono);
```

Otra función bastante útil en el manejo de ventanas, es **SDL_WM_IconifyWindow()** la cual simplemente hace que nuestra ventana sea minimizada y en caso se ejecutarse con éxito la aplicación recibirá un evento de pérdida **SDL_APPACTIVE**. El prototipo de esta función es el siguiente:

```
int SDL_WM_IconifyWindow(void);
```

Devuelve un valor distinto de cero en caso de éxito o un cero si se produjo algún error, el cual se puede deber a que la iconificación no está soportada o porque fue rechazada por el manejador de ventanas.

Por último veremos la función **SDL_WM_ToggleFullScreen()**, la cual intercambia entre modo ventana y modo a pantalla completa. Su prototipo es:

```
int SDL_WM_ToggleFullScreen(SDL_Surface *surface);
```

Como vemos el único parámetro que acepta es un puntero a una superficie, que frecuentemente será el puntero que nos devuelve la función **SDL_SetVideoMode()**, que nosotros llamamos **screen**. Esta función devuelve 0 en caso de fallo y 1 en caso de éxito.

Eventos

Muchos se preguntaran que son lo eventos. Bueno un evento como su nombre lo dice es un suceso que ocurre en nuestra aplicación, por ejemplo presionar un botón del mouse, presionar una tecla, cerrar la aplicación, etc. En general es una forma de permitir a la aplicación recibir entrada del usuario.

El manejo de eventos se inicializa junto con la inicialización del subsistema de video, es decir cuando realizamos la siguiente llamada de función:

```
SDL_Init(SDL_INIT_VIDEO);
```

Internamente SDL almacena todos los eventos sin procesar en un *cola de eventos*.

SDL trata a los eventos de una forma muy particular, a través de una unión **SDL_Event**, que veremos a continuación:

```
typedef union{
    Uint8 type;
    SDL_ActiveEvent active;
    SDL_KeyboardEvent key;
    SDL_MouseMotionEvent motion;
    SDL_MouseButtonEvent button;
    SDL_JoyAxisEvent jaxis;
    SDL_JoyBallEvent jball;
    SDL_JoyHatEvent jhat;
    SDL_JoyButtonEvent jbutton;
    SDL_ResizeEvent resize;
    SDL_QuitEvent quit;
    SDL_UserEvent user;
    SDL_SyWMEEvent syswm;
} SDL_Event;
```

Ahora veremos que significa cada uno de estos campos:

type: el tipo de evento.

active: evento de activación.

key: evento de teclado.

motion: evento de movimiento del mouse.

button: evento de botón del mouse.

jaxis: evento de movimiento de eje de joystick.

jball: evento de movimiento del trackball del joystick.

jhat: evento de movimiento del minijoystick (hat) del joystick.

jbutton: evento de botón del joystick.

resize: evento de redimensionado de ventana.

quit: evento de petición de cierre de aplicación.

user: evento definido por el usuario.

syswm: evento indefinido del gestor de ventanas.

Como podemos apreciar esta unión es un conjunto de todas las estructuras de eventos de la SDL, y su uso es una simple cuestión de conocimiento de los campos de cada una de las estructuras.

Es posible que nunca ocupemos todas estas estructuras. Las que mas se podrían utilizar son las del manejo del teclado y el mouse, pero para esto existen otras alternativas que veremos mas adelante.

Ahora veremos una lista de eventos soportados por SDL junto a la estructura que lo maneja.

Tipo de Evento	Estructura de Evento
SDL_ACTIVEEVENT	SDL_ActiveEvent
SDL_KEYDOWN/UP	SDL_KeyboardEvent
SDL_MOUSEMOTION	SDL_MouseMotionEvent
SDL_MOUSEBUTTONDOWN/UP	SDL_MouseButtonEvent
SDL_JOYAXISMOTION	SDL_JoyAxisEvent
SDL_JOYBALLMOTION	SDL_JoyBallEvent
SDL_JOYHATMOTION	SDL_JoyHatEvent
SDL_JOYBUTTONDOWN/UP	SDL_JoyButtonEvent
SDL_QUIT	SDL_QuitEvent
SDL_SYSWMEVENT	SDL_SysWMEvent
SDL_VIDEORESIZE	SDL_ResizeEvent
SDL_USEREVENT	SDL_UserEvent

La estructura `SDL_Event` tiene dos usos:

- Lectura de eventos de la cola de eventos.
- Inserción de eventos en la cola de eventos.

La lectura de eventos de la cola de eventos se realiza mediante la función **SDL_PollEvent()**. Esta función busca nuevos eventos pendientes en la cola. Su prototipo es el siguiente:

```
int SDL_PollEvent(SDL_Event *event);
```

Como vemos se le pasa un puntero a una unión `SDL_Event`. El evento obtenido es eliminado de la cola, si se le pasa `NULL` este no se elimina. Devuelve 1 si existe algún evento pendiente y 0 si no hay ninguno.

En todo programa que hagamos, vamos a querer manejar los eventos pendientes, por lo que la mayoría de la veces estaremos en un bucle recogiendo y procesando los eventos, esto lo podríamos hacer de la siguiente forma:

```
SDL_Event event;
```

```
.  
. while(SDL_PollEvent(&event))  
{  
  if(event.type==SDL_KEYDOWN) printf("Tecla presionada");  
}  
.  
.
```

Este ejemplo es simplemente para mostrar como manejar los eventos, aquí creamos una variable de tipo `SDL_Event`, luego dentro de un `while` verificamos si hay algún evento en la cola, si es así preguntamos de que tipo es, si se ha presionado algún tecla (`SDL_KEYDOWN`) imprimimos un mensaje.

En general el manejo de eventos en la SDL es muy sencillo, lo que puede hacerlo complicado es el alto numero de estructuras que existen.

Existen otras funciones, para insertar un evento en la cola, para esperar un cierto evento, etc. pero estas son poco usadas así que no las veremos.

Manejo del teclado

El manejo del teclado es muy sencillo en la SDL, pero puede hacerse de dos maneras, una es utilizando los eventos y la otra mediante la función **SDL_GetKeyState()**. Veamos primero esta última forma ya que es la más sencilla y no hay que estar trabajando con tantas estructuras. Lo que hace esta función es obtener una imagen del estado actual teclado, su prototipo es el siguiente:

```
Uint8 *SDL_GetKeyState(int *numkeys);
```

Esta función devuelve un puntero a un arreglo que contiene el estado de cada tecla. Un valor de 1 significa que la tecla esta presionada y 0 que no. El parámetro que toma corresponde al tamaño del arreglo, pero nosotros le pasaremos NULL.

El arreglo se encuentra indexado según los símbolos de tipo `SDLK_*`, por ejemplo las teclas de cursor son las siguientes: `SDLK_UP`, `SDLK_DOWN`, `SDLK_RIGHT`, `SDLK_LEFT`. Si quieren conocer el resto de los símbolos de las teclas pueden ver el apéndice A del curso.

Veamos ahora un ejemplo del uso de esta función. Supongamos que tenemos un personaje en nuestro juego y queremos moverlo al presionar las teclas de cursor. Aquí vamos a suponer que tenemos dos variables `x` e `y`, para almacenar la posición actual. Veamos como quedaría:

```
Uint8 *key;  
Uint16 x, y;  
.  
.  
key=SDL_GetKeyState(NULL);  
  
if(key[SDLK_UP])    y--;  
if(key[SDLK_DOWN]) y++;  
if(key[SDLK_RIGHT]) x++;  
if(key[SDLK_LEFT]) x--;  
.  
.
```

Veamos ahora el manejo de teclado mediante eventos. Para esto necesitamos conocer como está formada la estructura **SDL_KeyboardEvent**, veámosla:

```
typedef struct
{
    Uint8 type;
    Uint8 state;
    SDL_keysym keysym;
} SDL_KeyboardEvent;
```

Veamos ahora que significa cada campo:

type: aquí se guarda el evento ocurrido ya sea que se presionó o soltó una tecla, estos eventos son `SDL_KEYDOWN` o `SDL_KEYUP`.

state: igual que al anterior, indica el estado de la tecla. Los eventos son `SDL_PRESSED` o `SDL_RELEASED`.

keysym: otra estructura que almacena toda la información de la tecla presionada.

En la estructura **SDL_keysym**, es donde se almacena realmente cual fue la tecla que se presionó o soltó, veamos el contenido de esta estructura:

```
typedef struct
{
    Uint8 scancode;
    SDLKey sym;
    SDLMod mod;
    Uint16 unicode;
} SDL_keysym;
```

Para que quede más claro, veamos el significado de cada campo:

scancode: corresponde al código scan (scancode) el cual es dependiente del hardware.

sym: símbolo virtual de la tecla.

mod: modificadores actuales de la tecla.

unicode: carácter en formato unicode, siempre y cuando la traducción unicode este habilitada.

De estos campos el más importante y el que más se usará, es el campo `sym`, ya que contiene el símbolo de la tecla. Estos símbolos son los mismo que vimos anteriormente.

Ahora que ya conocemos como están formadas estas estructuras veamos el mismo ejemplo anterior, pero ahora manejado a través de eventos:

```
SDL_Event event;
Uint16 x, y;
.
.
while(SDL_PollEvent(&event))
{
  if(event.type==SDL_KEYDOWN)
  {
    switch(event.key.keysym.sym)
    {
      case SDLK_UP    : y--; break;
      case SDLK_DOWN : y++; break;
      case SDLK_RIGHT : x--; break;
      case SDLK_LEFT  : x++; break;
    }
  }
}
.
.
```

Como vemos esto no tiene mucha ciencia, lo que puede complicar es que existen varias estructuras. Bueno eso es todo sobre el manejo del teclado y no necesitarán saber nada más para crear sus aplicaciones o juegos.

Manejo del mouse

Al igual que con el manejo del teclado tenemos dos formas de manejar este periférico, ya sea con eventos o con la función **SDL_GetMouseState()**. Su prototipo es el siguiente:

```
Uint8 SDL_GetMouseState(int *x, int *y);
```

Esta función devuelve el estado de los botones como una máscara de bits que puede ser consultada mediante el uso de la macro **SDL_BUTTON(X)**, donde x puede tomar los valores 1, 2 o 3 que corresponden a los botones izquierdo, central y derecho respectivamente. Los parámetros x e y se establecen según la posición actual del puntero del mouse. Se puede utilizar NULL como valor para x e y si lo que se quiere es solo consultar, por el estado de los botones del mouse.

Veamos un ejemplo del uso de esta función:

```
Uint8 mouse_b;  
int mouse_x, mouse_y;  
  
SDL_GetMouseState(&mouse_x, &mouse_y);  
if((mouse_x >= 0) && (mouse_y >= 0) && (mouse_x <= 100) && (mouse_y <=100))  
{  
  mouse_b=SDL_GetMouseState(NULL, NULL)  
  if(mouse_b & SDL_BUTTON(1))  
  {  
    ...  
  }  
}
```

Lo que se hace en este ejemplo es verificar si el puntero del mouse se encuentra en un área determinada de la pantalla, si es así verificamos si se ha pulsado el botón izquierdo, y en caso de ser así hacemos algo.

Existe también una función para colocar el puntero del mouse en una posición determinada de la pantalla, esta es **SDL_WarpMouse()**, su prototipo es:

```
void SDL_WarpMouse(Uint16 x, Uint16 y);
```

La función es muy sencilla de utilizar, simplemente se la pasa como parámetro las nuevas coordenadas del puntero del mouse.

Por último otra función que nos puede ser útil, es **SDL_ShowCursor()**, la que activa o desactiva la visualización del cursor en la pantalla. Su prototipo es el siguiente:

```
int SDL_ShowCursor(int toggle);
```

Si toggle vale 0 el cursor será escondido de lo contrario si es 1 se mostrará. La función devuelve 1 si el cursor se estaba visualizando antes de la llamada, o 0 en caso contrario.

Ahora veremos el manejo del mouse mediante eventos. Para esto necesitaremos conocer dos estructuras, las que manejan los eventos del movimiento del puntero del mouse y el estado de los botones. Una de estas estructuras es **SDL_MouseMotionEvent**, veámosla más detalladamente:

```
typedef struct
{
    Uint8 type;
    Uint8 state;
    Uint16 x, y;
    Sint16 xrel, yrel;
} SDL_MouseMotionEvent;
```

Y como es tradición veamos el significado de cada campo:

type: indica el tipo de evento producido, este es SDL_MOUSEMOTION, es decir el movimiento del puntero del mouse.

state: el estado actual de los botones.

x, y: las coordenadas (x, y) del puntero del mouse.

xrel, yrel: movimiento relativo del puntero en la dirección (x, y).

La otra estructura es **SDL_MouseButtonEvent**, veámosla:

```
typedef struct
{
    Uint8 type;
    Uint8 button;
    Uint8 state;
    Uint16 x, y;
} SDL_MouseButtonEvent;
```

Y el significado de cada campo es:

type: el tipo de evento ocurrido, puede ser botón presionado o soltado, es decir: SDL_MOUSEBUTTONDOWN o SDL_MOUSEBUTTONUP.

button: el índice del botón del mouse, puede ser: SDL_BUTTON_LEFT, SDL_BUTTON_MIDDLE, SDL_BUTTON_RIGHT.

state: indica si se presionó o soltó un boton, es decir: SDL_PRESSED o SDL_RELEASED.

x, y: las coordenadas (x,y)del puntero del mouse, en el momento de la pulsación/liberación.

Veamos como quedaría nuestro ejemplo anterior del mouse, pero ahora utilizando eventos:

```
SDL_Event event;
int mouse_x, int mouse_y;
.
.
while(SDL_PollEvent(&event))
{
  if(event.type==SDL_MOUSEBUTTONDOWN)
  {
    if(event.button.button==SDL_BUTTON_LEFT)
    {
      mouse_x=event.button.x;
      mouse_y=event.button.y;
      if((mouse_x >= 0) && (mouse_y >= 0) && (mouse_x <= 100) && (mouse_y <=100))
      {
        ...
      }
    }
  }
}
.
.
```

Espero que haya quedado claro el manejo del mouse y de los eventos en general con la SDL. Con lo que vimos ya podemos crear cualquier tipo de aplicación, que utilice el teclado y/o el mouse.

Cómo cargar y mostrar imágenes en pantalla

Antes de mostrar una imagen en la pantalla, como es lógico, debemos cargarla en memoria con la función **SDL_LoadBMP()**, la cual carga imágenes en formato BMP a cualquier profundidad, en una superficie **SDL_Surface**. Veamos sus prototipo:

```
SDL_Surface *SDL_LoadBMP(const char *file);
```

El uso de esta función es muy sencilla, toma solo un parámetro el cual es una cadena que indica donde encontrar el archivo y devuelve un puntero a un superficie. Devuelve **NULL** si hubo algún error.

Otra función que nos puede ser útil, es **SDL_SaveBMP()**, que corresponde al complemento de la anterior. Esta función simplemente salva el contenido de una superficie en un archivo en formato BMP. Veamos sus prototipo:

```
int SDL_SaveBMP(SDL_Surface *surface, const char *file);
```

Esta función toma dos parámetros, el primero corresponde a la superficie que queremos guardar y el segundo el nombre del nuevo archivo. Devuelve 0 en caso de éxito o -1 si hubo algún error.

Una vez que tengamos cargada nuestra imagen en una superficie, lo común sería querer volcarla a la pantalla o a otra superficie, para esto utilizamos la función **SDL_BlitSurface()**, veamos su prototipo:

```
int SDL_BlitSurface(SDL_Surface*src, SDL_Rect *srcrect, SDL_Surface *dst,  
                    SDL_Rect *dstrect);
```

Lo que hace esta función es realizar un rápido volcado de la superficie origen a la superficie destino. Veamos ahora el significado de cada uno de sus parámetros, el primero corresponde a la superficie origen, es decir lo que queremos volcar. El segundo parámetro nos indica el área de la superficie origen que se copiará. El tercer parámetro es la superficie destino y por último el cuarto parámetro nos indica en que posición (x, y) se volcará la superficie, el ancho y el alto de **dstrect** son ignorados.

Pero existen unos casos especiales, si tanto **srcrect** como **dstrect** son **NULL**, entonces la superficie origen será volcada completamente en la superficie destino.

Esta función devuelve 0 si todo resultó correctamente y en cualquier otro caso devuelve -1.

Así como tenemos una función para cargar imágenes, existe otra para liberar la memoria ocupada por una superficie, esta función es **SDL_FreeSurface()**, veámosla:

```
void SDL_FreeSurface(SDL_Surface *surface);
```

Esta función es muy sencilla de utilizar, simplemente se le pasa un puntero a la superficie que queremos liberar.

Muchas veces cuando dibujamos en una superficie no queremos dibujar fuera de ella, una forma de solucionar esto es utilizar la función **SDL_SetClipRect()**, y lo que hace es establecer el rectángulo de recorte de una superficie, veámosla mas detalladamente:

```
void SDL_SetClipRect(SDL_Surface *surface, SDL_Rect *rect);
```

El primer parámetro es la superficie a la que queremos definirle un área de recorte y el segundo parámetro indica cual será esta área. Cuando esta superficie es el destino de un volcado, solo el área en el interior del rectángulo de recorte será pintada.

En caso de que el rectángulo rect sea más grande que el área de la superficie, este será recortado a los límites de la superficie, así que nunca caerá fuera de los límites.

Además si el parámetro rect es NULL, entonces el rectángulo de recorte se establecerá al tamaño completo de la superficie.

Existe otra función que nos puede ser muy útil algunas veces, ésta es **SDL_FillRect()**, y lo que hace es llenar alguna área de una superficie con un color determinado, veamos su prototipo:

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

El primer parámetro es la superficie que queremos llenar. El segundo parámetro corresponde al área de la superficie que llenaremos y por último tenemos el color. El color deberá ser un píxel con el formato utilizado por la superficie, y puede ser generado con la función **SDL_MapRGB()**.

Si dstrect es NULL, entonces la superficie completa será llenada con el color pasado. La función devuelve 0 en caso de éxito o -1 en caso de error.

Si se ha establecido un rectángulo de recorte en la superficie destino, esta función recortará basándose en la intersección del rectángulo de recorte y el rectángulo destino (dstrect).

Otra función muy útil es la **SDL_SetColorKey()**, la cual define el color transparente de una superficie, veamos su prototipo:

```
int SDL_SetColorKey(SDL_Surface *surface, Uint32 flag, Uint32 key);
```

El primero parámetro de esta función corresponde a la superficie en la que definiremos su color transparente. El segundo corresponde a unos flags que pueden ser los siguientes:

SDL_SRCCOLORKEY: indica que “key” será el valor del color transparente de la superficie.
SDL_RLEACCEL: indica que la superficie será pintada utilizando aceleración RLE, es decir con esto se puede incrementar sustancialmente la velocidad de volcado de imágenes con filas horizontales de pixeles transparentes (pixeles que concuerdan con el valor de la clave key).

Por último tenemos el tercer parámetro que es el valor del color que tomaremos como transparente en la superficie. La clave key deberá estar en el mismo formato de píxel de la superficie. A menudo se utilizará la función **SDL_MapRGB()** para obtener un valor aceptable. Además si el flag es cero, esta función limpia cualquier color clave.

La función devuelve 0 en caso de éxito, o -1 si se produjo algún error.

Frecuentemente utilizaremos los dos flags de esta forma:

```
SDL_SRCCOLORKEY | SDL_RLEACCEL
```

Como siempre buscamos realizar los volcados de superficies, en el menor tiempo posible, podemos utilizar la función **SDL_DisplayFormat()** para transformar una superficie en el formato de la pantalla, de esta forma al realizar los volcados a pantalla, estos se realizarán muy rápidamente. Veamos el prototipo de esta función:

```
SDL_Surface *SDL_DisplayFormat(SDL_Surface *surface);
```

Esta función es muy simple de utilizar, simplemente le pasamos la superficie que queremos transformar y ésta nos devolverá otra superficie con el formato que tiene el framebuffer de video.

Además si se desea aprovechar el volcado por color clave (colorkey) o el volcado alpha acelerados por hardware, deberemos establecer los valores de color clave y alpha antes de llamar a esta función.

Si la conversión falla o no hay memoria suficiente, la función devuelve NULL.

Ya que vimos todo lo necesario para manejar imágenes en la SDL, crearemos una función muy útil, que llamaremos **load_sprite()**. Esta función estará encargada de cargar desde un archivo una imagen, asignarle un color transparente que definiremos como el color magenta que tiene componentes (255, 0, 255), y por último transformar esta superficie al formato de pantalla. Veamos primero su prototipo:

```
SDL_Surface *load_sprite(const char *filename);
```

Como vemos es muy sencilla, y nos ahorrará bastante trabajo. Toma solo un parámetro y este corresponde al nombre del archivo de la imagen a cargar. Devuelve 0 si hubo algún error, o un puntero a la nueva superficie.

Veamos ahora su definición:

```
// Carga un sprite desde un archivo  
SDL_Surface *load_sprite(const char *filename)  
{  
    SDL_Surface *tmp, *bmp;  
    Uint32 color_key;  
  
    tmp=SDL_LoadBMP(filename);  
    if(!tmp) return 0;  
  
    color_key=SDL_MapRGB(tmp->format, 255, 0, 255);  
    SDL_SetColorKey(tmp, SDL_SRCCOLORKEY|SDL_RLEACCEL, color_key);  
    bmp=SDL_DisplayFormat(tmp);  
    SDL_FreeSurface(tmp);  
    if(!bmp) return 0;  
  
    return bmp;  
}
```

Y para terminar el tema de las imágenes veremos otras funciones más, las cuales nos ahorrarán trabajo cuando queramos comenzar a programar. Una de estas funciones que crearemos es la función **clear()**, la cual simplemente limpia o llena una superficie con un color determinado. Crearemos dos versiones de esta función, veamos primero sus prototipos:

```
void clear(SDL_Surface *screen);  
void clear(SDL_Surface *screen, SDL_Color color);
```

Como vemos aquí, se hace uso de la sobrecarga de funciones, ya que de esta forma es más fácil usarlas. La única diferencia entre estas funciones, es que la primera limpiará la superficie con el color por defecto que será el color 0, y la otra con un color pasado por parámetro.

Veamos ahora las definiciones de estas dos funciones:

```
// Limpia screen de color negro
void clear(SDL_Surface *screen)
{
    Uint32 col=SDL_MapRGB(screen->format, 0, 0, 0);
    SDL_FillRect(screen, 0, col);
}

// Limpia screen con el color especificado
void clear(SDL_Surface *screen, SDL_Color color)
{
    Uint32 col=SDL_MapRGB(screen->format, color.r, color.g, color.b);
    SDL_FillRect(screen, 0, col);
}
```

Y la función que nos falta es **draw_sprite()**, la cual como su nombre lo indica dibujará una imagen en pantalla. Veamos su prototipo:

```
void draw_sprite(SDL_Surface *screen, SDL_Surface *sprite, int x, int y);
```

La función toma cuatro parámetros, el primero de ellos corresponde a la superficie en donde dibujaremos la imagen, el segundo es la imagen en sí y por último tenemos la posición (x, y) en donde se dibujará.

Veamos su definición:

```
// Dibuja un sprite en screen
void draw_sprite(SDL_Surface *screen, SDL_Surface *sprite, int x, int y)
{
    SDL_Rect rect;

    rect.x=x;
    rect.y=y;

    SDL_BlitSurface(sprite, 0, screen, &rect);
}
```

Manejo del tiempo en la SDL

La librería SDL nos proporciona varias funciones para manejar el tiempo, es decir funciones para obtener el tiempo actual, esperar un intervalo de tiempo determinado, etc.

Una de las funciones que más usaremos al programar es **SDL_GetTicks()**, la cual devuelve el número de milisegundos desde que se inicializó la librería SDL. La función es muy sencilla de utilizar, veamos su prototipo:

```
Uint32 SDL_GetTicks(void);
```

Otra de las funciones útiles en el manejo del tiempo es **SDL_Delay()**, la cual espera un tiempo determinado de milisegundos, para después continuar con la ejecución del programa. Veámosla:

```
void SDL_Delay(Uint32 ms);
```

La función toma como parámetro un entero de 32 bits que indica el tiempo a esperar expresado en milisegundos. Hay que tener en cuenta que esta función espera al menos el tiempo especificado, pero es posible que tarde un poco más según las tareas que este procesando el sistema operativo, por lo tanto no es tan exacta.

La librería SDL nos permite agregar timers (temporizadores) para ejecutar una función callback cada cierto tiempo. La función que nos permite hacer esto es **SDL_AddTimer()**, veámosla:

```
SDL_TimerID SDL_AddTimer(Uint32 interval, SDL_NewTimerCallback callback,  
void *param);
```

SDL_NewTimerCallback corresponde al siguiente typedef:

```
typedef Uint32 (*SDL_NewTimerCallback)(Uint32 interval, void *param);
```

El primer parámetro nos indica el intervalo al que se ejecutará nuestra función, expresado en milisegundos. El segundo parámetro es la función callback que queremos que se ejecute a intervalos regulares de tiempo y por último tenemos los parámetros de la función callback.

La función callback es ejecutada en un hilo (thread) diferente al del hilo principal, así que no debe llamar desde la función callback del timer, a las funciones del programa principal en ejecución.

La máxima resolución de este timer es de 10 ms, es decir si se especifica un intervalo de 16 ms , la función callback tardará en ejecutarse aproximadamente 20 ms después que halla sido llamada.

La función devuelve el ID (identificador) del timer agregado o NULL si ha ocurrido un error durante la creación del timer.

Para eliminar un timer utilizamos la función **SDL_RemoveTimer()**, su prototipo es:

SDL_bool SDL_RemoveTimer(SDL_TimerID id);

A esta función se le debe pasar el identificador del timer a eliminar, el cual es obtenido con la función `SDL_AddTimer()`. Devuelve un valor booleano que indica si el proceso se llevó a cabo.

Por último para terminar con los timers, tenemos la función **SDL_SetTimer()**, la cual cambia el intervalo de tiempo en ejecución e incluso puede cambiar la función callback que ejecuta. Veámosla:

int SDL_SetTimer(Uint32 interval, SDL_TimerCallback callback);

SDL_TimerCallback es otro typedef:

typedef Uint32 (*SDL_TimerCallback)(Uint32 interval);

El primer parámetro será el nuevo intervalo de tiempo y el segundo parámetro la nueva función callback, en caso de querer cambiarla.

Una forma de cancelar un timer en ejecución es llamar a la función `SDL_SetTimer()` de la siguiente forma: **SDL_SetTimer(0, NULL);**

Para utilizar las funciones `SDL_AddTimer()`, `SDL_RemoveTimer()` y `SDL_SetTimer()` se debe llamar a la función `SDL_Init()` con el parámetro `SDL_INIT_TIMER`.

Creando nuestra propia librería

Como hemos visto hasta el momento, la SDL es bastante genérica, por lo que es conveniente que creamos nuestra propia librería, para así ahorrar trabajo a la hora de programar nuestros juegos o demos. Primero crearemos unas cuantas funciones que nos servirán mucho más adelante, ya hemos visto alguna de ellas. Además crearemos unas cuantas variables globales que podremos usar desde cualquier parte. Ahora veremos los prototipos de estas funciones y variables, que se encuentran en el archivo **util.h**, veámoslas:

```
#ifndef UTIL_H
#define UTIL_H

#include <SDL/SDL.h>

// Variables globales externas
extern SDL_Surface *screen;           // Pantalla
extern Uint8 *key;                   // Teclado
extern int SCREEN_W;                 // Ancho en pixeles de la pantalla
extern int SCREEN_H;                 // Alto en pixeles de la pantalla
extern int BPP;                       // Bit por pixel

// Prototipo de funciones
void showerror(char *msg);
void putpixel(SDL_Surface *screen, int x, int y, SDL_Color color);
SDL_Color getpixel(SDL_Surface *screen, int x, int y);
Uint32 get_pixel(SDL_Surface *screen, int x, int y);
void lock(SDL_Surface *screen);
void unlock(SDL_Surface *screen);
void draw_sprite(SDL_Surface *screen, SDL_Surface *sprite, int x, int y);
void clear(SDL_Surface *screen);
void clear(SDL_Surface *screen, SDL_Color color);
SDL_Surface *load_sprite(const char *filename);
SDL_Surface *load_sprite(const char *filename, SDL_Color color);

#endif
```

Estas funciones son bastantes sencillas, su implementación la pueden ver en el archivo **util.cpp** que se encuentra en el ejemplo de este capítulo.

Manejo de sprites y animaciones

Tal vez esta es una de las partes más esperadas, el manejo de sprites y animaciones. Bueno antes que todo, debemos saber que es un sprite. Un **sprite** básicamente es un objeto que podemos ver en la pantalla, que tiene asociado ciertos atributos, tales como posición, velocidad, etc., y por supuesto una o varias imágenes. Los sprites pueden **estáticos** o **dinámicos**. Los **estáticos** se podría decir que son aquellos objetos que están fijos en la pantalla, tales como un llama en el fondo de un juego, y los **dinámicos** aquellos que se mueven por la pantalla, como puede ser el personaje de nuestro juego favorito. Además los sprites pueden estar contruidos de una o más imágenes, estos últimos comúnmente son los que poseen animaciones, es decir una secuencia de varios **frames**, que vistos uno tras otro, a una velocidad determinada, dan el efecto de que se esta moviendo. Un **frame** es simplemente un cuadro de la animación, o dicho de otra forma una simple imagen que tiene asociada una posición, tiempo, etc.

Para manejar los sprites en nuestro juego crearemos dos clases, una encargada de la parte gráfica de este, es decir manejará las animaciones y sabrá como dibujarlas en pantalla, esta clase la llamaremos **CSpriteGraphic**. La otra estará encargada de la parte lógica del sprite, es decir, que hacer en caso de que suceda algún evento, en que momento mostrar una u otra animación, el movimiento de este, etc. Esta última clase se llamará simplemente **CSprite**.

Veamos ahora el prototipo de la clase **CSpriteGraphic**:

```
#ifndef CSPRITE_GRAPHIC_H
#define CSPRITE_GRAPHIC_H

#include <SDL/SDL.h>
#include <vector>

using namespace std;

class CSpriteGraphic
{
public:
    CSpriteGraphic();
    ~CSpriteGraphic();
    int init(const char *dir, int num_frames, int time, const char *ext="pcx");
    int init(const char *dir, int num_frames, int time, SDL_Color color,
            const char *ext="pcx");
    void add_frame(SDL_Surface *image, int cx, int cy, int time);
    void draw(SDL_Surface *buffer, int x, int y, int num);
    int get_num_frames();
    int get_time(int num=0);
    int get_w(int num=0);
    int get_h(int num=0);
    SDL_Surface *get_frame(int num=0);

private:
    struct SpriteFrame
    {
        SDL_Rect center;
        SDL_Surface *image;
        int time;
    };

    vector <SpriteFrame> frames;
};

#endif
```

Veamos ahora la siguiente clase, la **CSprite**, dentro de esta tenemos una referencia a un objeto de tipo CSpriteGraphic:

```
#ifndef CSPRITE_H
#define CSPRITE_H

#include <SDL/SDL.h>
#include "CSpriteGraphic.h"

class CSprite
{
public:
    CSprite(CSpriteGraphic *graph=NULL);
    virtual ~CSprite();
    void init_graph(CSpriteGraphic *graph);
    int load_anim(const char *filename);
    virtual void update();
    void draw(SDL_Surface *buffer);
    void draw(SDL_Surface *buffer, int num_frame);
    void animate();
    void set_actual_frame(int num);
    int get_actual_frame();
    void start_anim();
    void stop_anim();
    void toggle_anim();
    void rewind();
    void set_pos_x(int x);
    void set_pos_y(int y);
    int get_pos_x();
    int get_pos_y();
    int get_w(int num=0);
    int get_h(int num=0);
    void set_actual_anim(int num);
    int get_actual_anim();
    bool get_full_anim();

protected:
    SDL_Rect pos;

private:
    CSpriteGraphic *graph;
    int actual_frame;
    int last_time;
    bool anim;
    bool full_anim;
    int actual_anim;
    int actual_secuencia;
    typedef vector <int> secuencia_t;
    vector <secuencia_t> animation;
};

#endif
```

La implementación de las clases `CSprite` y `CSpriteGraphic`, las podrán encontrar en el ejemplo de este capítulo.

Ahora cuando queramos crear algún objeto en nuestra pantalla, lo primero que haremos será crear un objeto de tipo `CSpriteGraphic` y cargar sus gráficos. Luego crear otro objeto de tipo `CSprite` y a este último asignarle el objeto de gráficos anteriormente creado. Cabe destacar, que para el manejo de las animaciones se hace uso de un archivo de texto que contiene la secuencia de cada una de las animaciones, la estructura de este archivo es super simple. La primera línea contiene un entero con el número total de animaciones. Luego por cada línea irá una secuencia de números (enteros) que contienen el número del frame a mostrar en esa animación, cada animación termina con el número `-1`. Para que quede más claro veamos el archivo `tux.ani` del ejemplo de este capítulo:

Archivo tux.ani

```
2
0 1 2 3 4 5 6 7 8 9 -1
10 11 12 13 14 15 16 17 18 19 -1
```

La clase CGame

Ya que tenemos la forma de manejar lo más básico en un juego, es decir los sprites, ahora crearemos una clase que contenga todo los objetos de un juego determinado. Esta clase la llamaremos **CGame**, y estará encargada de inicializar la SDL y un modo gráfico. Veamos el prototipo de esta clase, que es bastante simple:

```
#ifndef CGAME_H
#define CGAME_H

#include <SDL/SDL.h>

class CGame
{
public:
    CGame();
    virtual ~CGame();
    void init_graph(int w, int h, int bpp, const char *name, bool fullscreen=false);
    virtual void init();
    virtual void shutdown();
    virtual void main();
};

#endif
```

Al tener esta clase ya nos podemos olvidar de cómo inicializar la SDL y un modo gráfico, ya que ésta lo hace por nosotros. Por lo tanto en cada juego que creamos existirá una clase derivada de CGame.

Resumen

Este capítulo ha sido muy largo e intenso, pero ha valido la pena ya que disponemos de una pequeña librería con clases, para hacer más fácil nuestro trabajo en la construcción de algún juego.

En este capítulo vimos el manejo de ventanas y de eventos, también el manejo de dos periféricos muy importantes en cualquier juego el teclado y el mouse. Además vimos como cargar y mostrar imágenes en pantalla y el manejo del tiempo en nuestros programas. Por último construimos una pequeña librería que nos ahorrará trabajo más adelante, junto a un set de clases que irá creciendo a medida que avancemos en el curso.

A este capítulo lo acompaña un ejemplo de lo que podría ser un juego. Se trata del movimiento de un personaje en la pantalla, el cual puede moverse en dos direcciones (izquierda y derecha), correr y saltar. Además se incluye un efecto de nieve.

El contenido del próximo capítulo aun no está bien definido, pero algunas alternativas pueden ser el aprender a colocar texto en la pantalla y reproducción de sonidos/música en nuestros programas.