Introducción a SDL
(Simple Directmedia Layer)

Javier Martínez Baena (jbaena@decsai.ugr.es)


Universidad de Granada


Dpto. Ciencias de la Computación
e Inteligencia Artificial

Índice

1. Introducción

SDL es una biblioteca multiplataforma (Linux, Windows, ...) para el control multimedia del ordenador. Entre otras cosas permite controlar los sistemas de vídeo y audio y nos da la posibilidad de gestionar los eventos del sistema (pulsaciones de teclas, movimientos de ratón o joystick, etc).

En la página web de la SDL tienes enlaces a multitud de aplicaciones que han sido construidas haciendo uso de ella. Por ejemplo, la versión de Linux del juego *"Civilization: Call to Power"* (`http://www.lokigames.com/products/civctp/`):

Este documento únicamente pretende servir de apoyo a la documentación oficial de la SDL y por tanto será necesario consultar dicha documentación para determinadas cuestiones.

2. Compilación y enlazado

Para poder crear programas que hagan uso de SDL necesitamos tener instalado el paquete para desarrollo de esta biblioteca: `sdl-devel`. Una vez instalado hemos de saber que los ficheros de cabecera están en `/usr/include/SDL`. El fichero que hemos de incluir en nuestros programas se llama `SDL.h`. La biblioteca se encuentra en `/usr/lib` y se llama `libSDL.a`.

Por ejemplo, supongamos que nuestro fichero fuente es `ejemplo.cpp`:

```
1 // Fichero: ejemplo.cpp
2 #include "SDL.h"
3 ...
```

Para compilar usaremos la instrucción:

```
1 g++ -c ejemplo.cpp -I/usr/include/SDL
```

Para enlazar usaremos la instrucción:

```
1 g++ -o ejemplo ejemplo.o -lSDL -L/usr/lib
```

aunque la opción `-L/usr/lib` no es estrictamente necesaria ya que, al ser uno de los directorios estándar del sistema, el compilador buscará ahí, por defecto, las bibliotecas.

3. Uso de SDL

3.1. Inicialización de SDL

Antes de poder usar ninguna función de la SDL es necesario inicializarla. La SDL se compone de distintos subsistemas: uno para el control del vídeo, otro para el sonido, etc. Al inicializar la SDL hay que indicarle cuales de esos módulos necesitamos usar. Por ejemplo, si vamos a usar el vídeo pondremos:

```
1 SDL_Init(SDL_INIT_VIDEO);
```

Si queremos inicializar el vídeo y el audio pondremos:

```
1 SDL_Init(SDL_INIT_VIDEO|SDL_INIT_AUDIO);
```

Esta función devuelve `0` si todo es correcto y `-1` si hay algún error.

3.2. Gestión de errores con SDL

Cuando se produce algún error podemos usar la función `SDL_GetError`. Nos mostrará un mensaje indicando las causas del mismo:

```
cout << SDL_GetError();
```

3.3. Finalización de SDL

Antes de finalizar el programa debemos indicarle a la SDL que vamos a finalizar el uso de la SDL. Lo haremos con la instrucción:

```
SDL_Quit();
```

Para evitar tener que poner llamadas a la función `SQL_Quit()` en todos los puntos de salida del programa podemos usar la función `atexit()` justo tras la inicialización de la SDL de la siguiente forma:

```
atexit(SDL_Quit);
```

Con esto le estamos indicando que, en caso de finalizar nuestro programa no olvide ejecutar la función `SDL_Quit()` para liberar los recursos que hayan sido ocupados.

Ejemplo

```
#include "SDL.h"
...
int main(int argc, char *argv[])
{
  if (SDL_Init(SDL_INIT_VIDEO)<0) {
    cerr << "Hubo un error al iniciar el sistema de video" << endl;
    cerr << SDL_GetError() << endl;
    exit(1);
  }
  ...
    < aquí va mi programa >
  ...
  SQD_Quit();
}
```

4. Creando ventanas

Para crear una ventana usaremos la instrucción `SDL_SetVideoMode()` indicando las dimensiones de la ventana y la profundidad de color que deseamos (número de bits que vamos a usar para representar cada pixel). Además se podrán especificar algunos otros parámetros. Lo mas sencillo es usar la llamada de la siguiente forma:

```
...
SDL_Surface *pantalla;
pantalla=SDL_SetVideoMode(ancho,alto,bpp,SDL_SWSURFACE|SDL_ANYFORMAT)
if (pantalla==0) {
  cerr << "Error al crear ventana" << endl;
  cerr << SDL_GetError() << endl;
  SDL_Quit();
  exit(1);
}
...
```

Esta función devuelve un puntero a una variable de tipo `SDL_Surface` que representa nuestra ventana. Esta variable, entre otras cosas, contiene las dimensiones que hemos asignado a la ventana y un puntero a lo que podríamos considerar nuestra *pantalla virtual*.

El control del sistema de vídeo se realiza por parte de X-Windows. SDL es una biblioteca que nos alivia la tarea de gestionar aplicaciones gráficas (la programación con X-Windows es muy laboriosa y además sólo tiene sentido usarla con sistemas de tipo Unix ya que Windows tiene su propio entorno gráfico distinto a X-Windows). Para simplificarnos la tarea, SDL nos proporciona lo que podríamos considerar como un lienzo en el que nosotros podremos trabajar (dibujar): el framebuffer. Ese lienzo no es mas que una matriz bidimensional con las dimensiones de la ventana física con la que se corresponde de forma que cada pixel

ocupa una serie de bits para representar su color (profundidad de color o bits por pixel -bpp-). El tamaño de esa matriz será, en bits, (ancho x alto x bpp). Virtualmente podemos considerar que ese lienzo es la ventana en la que estamos trabajando.

Por defecto, cuando dibujamos en este lienzo, la información que vemos en el monitor no se ve alterada. En realidad estamos modificando la matriz bidimensional del lienzo (que está en memoria RAM) pero no la memoria de vídeo de nuestra tarjeta gráfica. Para indicarle al sistema gráfico que actualice la información que vemos en el monitor usaremos la instrucción `SDL_UpdateRect`.

Una vez que hemos creado la ventana podemos usar la función `SDL_WM_SetCaption()` para ponerle un título (ver el ejemplo de la sección 8).

5. Dibujando

El modelo de color que usan los monitores suele ser RGB. En este modelo cada pixel de la imagen se forma mezclando tres componentes básicos de color (Rojo, Verde y Azul) y por tanto para representar el color de un pixel lo que se hace es indicar que cantidad de cada uno de esos colores hemos de usar en la mezcla. Si usásemos una profundidad de color de 24 bpp podríamos considerar que los 8 primeros bits (1 byte) indican la cantidad de rojo, los 8 siguientes la cantidad de verde y los 8 restantes la cantidad de azul. Pero podemos usar otras profundidades de color. Si nuestra profundidad de color fuese 8 tendríamos que repartir esos bits entre las tres componentes, por ejemplo podríamos usar 2 bits para el rojo, 3 para el verde y 3 para el azul. De esta forma el valor de rojo será un valor entre 0 y 3 y los valores de verde y azul estarán entre 0 y 7.

Los bits por pixel que podemos usar suelen ser alguno de los siguientes valores: 8, 16, 24, 32. Es posible que nuestro entorno gráfico no permita trabajar con la profundidad de color que nosotros le pedimos a la SDL. En ese caso, la SDL tiene dos opciones: dar un error o simular que el entorno gráfico trabaja con esa profundidad de color. Para que se comporte de la segunda forma usamos el parámetro `SDL_SWSURFACE | SDL_ANYFORMAT`. La SDL se encargará de transformar nuestros dibujos a la profundidad de color adecuada que actualmente esté funcionando en nuestro ordenador.

`SDL_Surface` es un `struct` que tiene, entre otros, los siguientes campos:

```
int w; // Ancho de la pantalla
int h; // Alto de la pantalla
void *pixels; // Puntero al lienzo virtual
Uint16 pitch; // Tamaño, en bytes, que ocupa una línea del lienzo
              // (Uint16 es un entero que ocupa 2 bytes de memoria)
SDL_PixelFormat *format; // struct que contiene información sobre

                         // el formato de los píxeles de este "lienzo"
```

El `struct SDL_PixelFormat` contiene, entre otros los siguientes campos:

```
Uint8 BytesPerPixel;  // Número de bytes que ocupa un pixel
Uint8 BitsPerPixel;   // Número de bits que ocupa un pixel
```

`pitch` se calcula como `w*bpp/8`, es decir, el número de píxeles de ancho por el número de bytes que ocupa cada pixel. Por ejemplo, para averiguar la posición dentro del vector `*pixel`, del punto que está en la coordenada `(X=2, Y=5)`, debemos hacer el siguiente cálculo: `posi=Y*pitch+X*bpp`.

Cada vez que vamos a dibujar un punto debemos calcular la posición que ocupa el punto dentro del lienzo y separar los bits que codifican las componentes RGB del pixel para trabajar con ellas de forma independiente. Esta tarea es laboriosa y por tanto podemos crear una función que se encargue de recibir las coordenadas (x,y) en donde queremos pintar y el color (r,g,b) que queremos para el punto. Los valores de R, G y B deberían estar de acuerdo con la profundidad de color pero de nuevo esto supone que, dependiendo del bpp, tendríamos que dar unos valores de R,G,B u otros. Por ejemplo, si optamos por tener bpp=24 y queremos pintar en blanco debemos dar como color (R=255, G=255, B=255); en cambio, si optásemos por bpp=8 (según el ejemplo anterior), el blanco sería (R=3, G=7, B=7). Esto supone que nuestras aplicaciones se van a complicar bastante ya que tendrían que preveer la posibilidad de trabajar con cualquier profundidad de color. Para solventar este problema, la SDL nos proporciona una función llamada `SDL_MapRGB`. Esta función recibe un color en formato RGB (usando un byte para cada componente) y nos devuelve su equivalente usando los bpp de la pantalla en la que estamos trabajando. Devuelve un número entero de 32 bits (que sería la profundidad máxima de color permitida) en el que ya está codificado el color de acuerdo a la profundidad de color de la pantalla.

El siguiente ejemplo muestra una función que pinta en la coordenada (x,y) un punto de color (r,g,b):

```

// Dibujamos un punto de color (r,g,b) en la posicion (x,y) en la pantalla
// Esta función está adaptada partiendo de la que aparece en el manual de SDL
void PutPixel(SDL_Surface *pantalla, int x, int y,
              unsigned char r, unsigned char g, unsigned char b)
```

```
{
  // Bytes que ocupa un pixel
  int bpp = pantalla->format->BytesPerPixel;

  // p es un puntero al pixel que vamos a pintar
  Uint8 *p = (Uint8*)pantalla->pixels + y*pantalla->pitch + x*bpp;

  // color contendrá el valor codificado del color con el que vamos a pintar el pixel
  // teniendo ya en cuenta la profundidad de color
  Uint32 color = SDL_MapRGB(pantalla->format,r,g,b);

  switch (bpp) {
    case 1:
       *p=color;
       break;
    case 2:
      *(Uint16*)p = color;
      break;
    case 3:
     // Dependiendo del Sistema Operativo o la arquitectura hardware de nuestro ordenador
     // podemos tener distintas representaciones de los números (que ocupan mas de un byte)
     // en la memoria. En unos casos se almacena primero el byte menos signiticativo del
     // número (los bytes que forman la representación se escriben en orden inverso) y en
     // otros casos el primer bytes es el más significativo (los bytes mantienen el orden
     // en la representación interna). Los primeros se conocen como Little-Endian y los
     // segundos como Big-Endian.
     if (SDL_BYTEORDER==SDL_BIG_ENDIAN) {
        p[0]=(color»16)&0xff;
        p[1]=(color»8)&0xff;
        p[2]=color&0xff;
      } else {
        p[0]=color&0xff;
        p[1]=(color»8)&0xff;
        p[2]=(color»16)&0xff;
      }
      break;
    case 4:
      *(Uint32*)p = color;
      break;
  }
}
```

De forma análoga podemos obtener el color de un pixel del lienzo con la siguiente función:

```
// Obtenemos el color del punto que hay en las coordenadas (x,y)
// Devolvemos el valor en (r,g,b)
void GetPixel(SDL_Surface *pantalla, int x, int y,
              unsigned char &r, unsigned char &g, unsigned char &b)
{
  int bpp = pantalla->format->BytesPerPixel;
  Uint8 *p = (Uint8*)pantalla->pixels + y*pantalla->pitch + x*bpp;
  Uint32 color=0;

  switch (bpp) {
    case 1:
      color = *p;
      break;
    case 2:
      color = *(Uint16*)p;
      break;
    case 3:
      if (SDL_BYTEORDER==SDL_BIG_ENDIAN)
        color = p[0]«16 | p[1]«8 | p[2];
      else
```

```
        color = p[0] | p[1]«8 | p[2]«16;
      break;
    case 4:
      color = *(Uint32*)p;
      break;
  }

  // Decodificamos el color en sus 3 componentes R, G y B
  SDL_GetRGB(color,pantalla->format,&r,&g,&b);
}
```

En este caso hacemos uso de la función `SDL_GetRGB()` a la que le damos el color de un pixel (en el formato correspondiente según los bpp) y nos devuelve con que valores R, G y B se corresponde.

6. Lock y Unlock

Finalmente queda un detalle a tener en cuenta. Para poder tener acceso directo a la matriz a la que hemos llamado lienzo y modificarla a nuestro antojo primero hay que indicarle a la SDL que lo vamos a hacer (`SDL_LockSurface`). Una vez que hemos terminado de modificar dicha matriz le decimos a la SDL que hemos terminado de modificarla (`SDL_UnlockSurface`). Estas operaciones de bloqueo y desbloqueo no siempre son necesarias (depende del tipo de pantalla y otras cuestiones) y para saber cuando tenemos que hacerlas o no podemos usar la función `SDL_MUSTLOCK`.

`SDL_LockSurface` devuelve `0` si la operación se ha realizado con éxito y `-1` si hay algún error.

Si `SDL_MUSTLOCK` devuelve `0` significa que la pantalla no necesita realizar las operaciones de bloque o desbloqueo. Otro valor significará que necesitamos realizar el bloqueo y el desbloqueo.

Por tanto el esquema para dibujar es el siguiente:

```
if (SDL_MUSTLOCK(pantalla)) {
  if (SDL_LockSurface(pantalla)<0) {
    cerr << "Hay un error desbloqueando la pantalla" << endl;
    cerr << SDL_GetError() << endl;
    SDL_Quit();
    exit(1);
  }
}
...
... < dibujamos pixeles en el lienzo > ...
...
if (SDL_MUSTLOCK(pantalla)) {
  SDL_UnLockSurface(pantalla)
}
```

En la sección 4 hemos visto que tras hacer cualquier modificación en el lienzo es necesario indicarle a la SDL que sincronice la información que aparece en pantalla con la que tenemos en nuestro lienzo. Esta operación de actualización se hará después de haber ejecutado la instrucción SDL_UnLockSurface (si es que hubiese sido necesaria). En concreto la instrucción es `SDL_UpdateRect`. Seguramente no habremos hecho una modificación completa del lienzo (de todos sus píxeles) por lo que no será necesario pintar la ventana completa: sólo tiene sentido (por cuestiones de eficiencia) que actualicemos en pantalla aquella parte del lienzo que ha sido modificada. Con la función `SDL_UpdateRect` podemos indicar que rectángulo del lienzo queremos actualizar. La función lleva 4 parámetros (además del que indica la pantalla sobre la que estamos trabajando): `x, y, w, h`.

`(x,y)` son las coordenadas de la esquina superior izquierda del rectángulo que vamos a actualizar.

`w` : ancho del rectángulo.

`h` : altura del rectángulo.

Si, por ejemplo, nuestra ventana tiene un tamaño de 800x600 y sólo hemos pintado en la mitad superior de la misma llamaremos a la función:

```
SDL_UpdateRect(pantalla,0,0,800,300);  // x=0, y=0, w=800, h=300
```

Si lo que queremos es actualizar la ventana completa sin importarnos sus dimensiones lo haremos así:

```
SDL_UpdateRect(pantalla,0,0,0,0);
```

7. Gestión de eventos

Con lo que hemos visto hasta ahora podemos crear aplicaciones gráficas pero aún nos queda un problema muy importante por resolver. Si hacemos un programa que haga un dibujo usando los conocimientos que se han explicado con anterioridad nos deleitaremos viendo nuestra creación pero tras minimizar, maximizar o superponer alguna otra ventana sobre la nuestra veremos que algo no va bien: la ventana no se redibuja cuando queda total o parcialmente oculta por otras ventanas o cuando es minimizada y después maximizada.

Los entornos gráficos (Windows, X-Windows, etc) son los encargados de realizar este tipo de operaciones (actualización de los contenidos de las ventanas) pero sólo lo hacen cuando la aplicación concreta lo solicita. Es decir, si nosotros no le decimos a X-Windows que actualice la ventana tras haberla maximizado, este no hará nada (y veremos la ventana sin nada). Pero claro, ¿como podemos saber nosotros desde nuestra aplicación que la ventana ha cambiado sus condiciones de visualización?

Para contestar a esta pregunta y solucionar el problema, los entornos gráficos funcionan de la siguiente forma. El entorno es el encargado de procesar las pulsaciones de teclas, los movimientos de ratón, la minimización de una ventana, el cambio de tamaño de una ventana, etc. Es lógico que esto no lo hagan por su cuenta las aplicaciones ya que en un momento dado podemos tener varias funcionando a la vez (los sistemas actuales son multitarea) y si todas se encargasen de procesar, por ejemplo, las pulsaciones del teclado, se montaría un gran lío ya que todas las aplicaciones procesarían todas las pulsaciones (cuando lo lógico es que las pulsaciones sean analizadas sólo por la aplicación activa en cada momento). Cuando ocurre alguna de estas cosas (*evento*) el sistema gráfico se la envía sólo a la aplicación activa (y no al resto). Para hacer esto el sistema mantiene una *cola de eventos* para cada aplicación. A esa cola se pueden enviar eventos del tipo *"Se ha pulsado la tecla X"*, *"El ratón se ha movido y sus nuevas coordenadas son 12,45"*, *"La ventana de la aplicación ha sido minimizada"*, *"Ha cambiado el tamaño de la ventana a 200x400"*, etc.

Cada aplicación es consciente de que esa cola existe y, además, ella es la encargada de decidir que quiere hacer cuando recibe cada uno de esos eventos. Por ejemplo, si a la aplicación le llega el evento *"La ventana ha pasado a primer plano"*, es decir, la ventana estaba en segundo plano (minimizada o semioculta por otra ventana) y el usuario la ha seleccionado para trabajar en ella (ha sido maximizada o se ha ocultado la ventana que la ocultaba total o parcialmente) lo que debe hacer la aplicación es, por ejemplo, decirle al entorno gráfico que vuelva a redibujar la ventana con los contenidos que tenía antes de ser ocultada.

La SDL nos proporciona varias funciones y estructuras de datos para gestionar los eventos. Una de las funciones más importantes es `SDL_PollEvent()`, cuyo objetivo es examinar esa cola de eventos y, si hay alguno, informar a nuestro programa de que dicho evento existe, dándonos opción a hacer algo al respecto.

Los eventos están agrupados por categorías (teclado, ratón, ventanas, joystick, etc) y para cada uno de esos tipos existe un `struct` que almacena la información correspondiente al evento. Además, existe un tipo `union` que es `SDL_Event` para agrupar todos esos `struct` en un único tipo de dato. Algunos de los campos de esta `union` son:

```
typedef union {
  Uint8 type;   // Tipo de evento
  SDL_KeyboardEvent key;   // Si el evento tiene que ver con el teclado aquí
                           //está la información asociada
  SDL_MouseMotionEvent motion; // Para eventos de movimiento del ratón
  SDL_MouseButtonEvent button; // Para eventos de pulsación del ratón
  SDL_QuitEvent quit;  // Cuando el evento es de finalización dela aplicación
  ...
} SDL_Event;
```

Sabiendo esto, nuestra aplicación deberá consistir en un bucle que se dedique continuamente a explorar la cola de eventos y a actuar en consecuencia. Mientras estemos haciendo cálculos y no saquemos eventos de esa cola el entorno gráfico no hará nada con nuestra ventana.

Por ejemplo, tras hacer un dibujo deberíamos tener un bucle similar al siguiente para evitar que la pantalla se quede sin actualizar:

```
SDL_Event mi_evento;  // Aquí almacenaremos los eventos que vamos sacando de la cola
while (true) { // Bucle infinito (no acabamos nunca)
  while (SDL_PollEvent(&mi_evento)) {  // Miramos y sacamos el siguiente evento en cola
    // Aquí podríamos hacer algo para procesar los eventos
  }
}
```

La función `SDL_PollEvent()` devuelve `1` si hay algún evento pendiente en cola. En ese caso nos devuelve un puntero a dicho evento a través del parámetro y lo saca de la cola. Si no hay eventos en cola devuelve `0`.

Si, por ejemplo, deseamos que nuestra aplicación acabe al pulsar la tecla *Escape* o al pulsar la X para cerrar la ventana podríamos usar el siguiente bucle:

```
SDL_Event mi_evento;  // Aquí almacenaremos los eventos que vamos sacando de la cola
bool fin=false;       // Con esto controlamos cuando queremos acabar el programa
while (!fin) { // Bucle infinito (no acabamos nunca)
  while (SDL_PollEvent(&mi_evento)) {  // Miramos y sacamos el siguiente evento en cola
    switch (mi_evento.type) {   // Comprobamos el tipo de evento
      case SDL_KEYDOWN :   // Si es una pulsación de tecla
        if (evento.key.keysym.sym==SDLK_ESCAPE)   // Si es la tecla Escape
          fin=true;   // Ponemos fin a true para salir del bucle
        break;
      case SDL_QUIT : // Si hemos pulsado el cierre de la ventana
        fin=true;     // También acabamos
        break;
    }
  }
}
```

Algunos ejemplos de tipos de eventos son los siguientes:

- `SDL_KEYDOWN` : Cuando se pulsa una tecla.
- `SDL_KEYUP` : Cuando se suelta una tecla.

  Cuando ocurre alguno de estos dos eventos la información se guarda en un `struct SDL_KeyboardEvent` llamado `key` (dentro de `SDL_Event`). Este `struct` contiene, por ejemplo, un campo llamado `state` que puede valer `SDL_PRESSED` o `SDL_RELEASED` y que nos indica si la tecla ha sido pulsada o soltada. También tiene otro campo que es un nuevo `struct SDL_keysym` en donde hay información concreta sobre la tecla que se ha pulsado. De forma análoga existen diferentes `struct` para cada uno del resto de tipos de eventos.
- `SDL_MOUSEMOTION` : Cuando se mueve el ratón.
- `SDL_MOUSEBUTTONDOWN` : Cuando pulsamos un botón del ratón.
- `SDL_MOUSEBUTTONUP` : Cuando soltamos un botón del ratón.
- `SDL_QUIT` : Cuando pulsamos el cierre de la ventana.
- `SDL_ACTIVEEVENT` : Cuando la ventana obtiene o pierde el control del teclado o ratón (foco).
- `SDL_VIDEORESIZE` : La ventana ha cambiado su tamaño.

8. Ejemplo

El ejemplo que viene a continuación crea una ventana y dibuja varias cosas. Cuando movemos el ratón por la pantalla nos muestra en consola las coordenadas y el valor del pixel en formato RGB.

```
// Para compilar y enlazar:
//   g++ -o ejemplosdl ejemplosdl.cpp -I/usr/include/SDL -lSDL
// Necesitas instalar los paquetes:
//   SDL
//   SDL-devel

#include <iostream>
#include "SDL.h"

using namespace std;

#include "PutPixel.cpp"
#include "GetPixel.cpp"

void Dibujar(SDL_Surface *pantalla)
{
  // Dibujamos una línea roja en la esquina superior izquierda de la ventana
  for (int f=0; f<256; f++)
    PutPixel(pantalla,f,f,255,0,0);
```

```
16    // Dibujamos otra línea amarilla
17    for (int f=0; f<256; f++)
18      PutPixel(pantalla,256-f,f,255,255,0);

19    // Dibujamos colores aleatorios al lado de las líneas en cruz
20    srand(time(0));
21    for (int f=0; f<256; f++)
22      for (int c=256; c<512; c++)
23        PutPixel(pantalla,c,f,rand()%256,rand()%256,rand()%256);

24    // Dibujamos colores con componente R=255 debajo de lo anterior
25    for (int f=256; f<512; f++)
26      for (int c=0; c<256; c++)
27        PutPixel(pantalla,c,f,255,f-256,c);

28    // Dibujamos colores con la componente B=255 al lado del anterior
29    for (int f=256; f<512; f++)
30      for (int c=256; c<512; c++)
31        PutPixel(pantalla,c,f,f-256,c-256,255);
32  }

33  void ProcesaEventos(SDL_Surface *pantalla)
34  {
35    SDL_Event evento;    // Variable que almacena el evento que se ha producido
36    bool fin=false;      // Controlamos el final del programa
37    while (!fin) {
38      // Comprobamos si hay algún evento disponible esperando para ser procesado
39      // SDL_PollEvent devuelve 1 si existe algún evento y lo devuelve en "evento"
40      //                        0 si no existe ningún evento por procesar
41      while (SDL_PollEvent(&evento)) {

42        // Dependiendo del evento haremos una cosa u otra
43        switch (evento.type) {

44          // Si movemos el ratón mostramos información del punto
45          case SDL_MOUSEMOTION:
46            int x,y;
47            unsigned char r,g,b;

48            // Coordenadas del ratón
49            x = evento.motion.x;
50            y = evento.motion.y;

51            // Bloqueamos la pantalla si es necesario
52            if (SDL_MUSTLOCK(pantalla)) {
53              if (SDL_LockSurface(pantalla)<0) {
54                cerr«"Error al bloquear pantalla ("«SDL_GetError()«")"«endl;
55                SDL_Quit();
56                exit(-1);
57              }
58            }

59            // Obtenemos el color del pixel
60            GetPixel(pantalla,x,y,r,g,b);

61            // Desbloqueamos la pantalla si es necesario
62            if (SDL_MUSTLOCK(pantalla)) {
63              SDL_UnlockSurface(pantalla);
64            }

65            // Mostramos la información en consola
66            cout « "(X,Y)=("«x«","«y«")      (R,G,B)=("«
67                                 (int)r«","«(int)g«","«(int)b«")"«endl;
68            break;
```

```
69          case SDL_KEYDOWN:
70            // Si pulsamos Escape acabamos
71            if (evento.key.keysym.sym==SDLK_ESCAPE) {
72              cout « "Has pulsado escape. Hasta otra !"«endl;
73              fin=true;
74            }
75            break;

76          case SDL_QUIT:
77            // Si cerramos la ventana ...
78            cout « "Has cerrado la ventana. Hasta otra !"«endl;
79            fin=true;
80            break;
81        }
82      }
83    }
84  }

85  int main(int argc, char *argv[])
86  {
87    // Variable que representa la pantalla donde vamos a dibujar
88    SDL_Surface *pantalla;

89    // Iniciamos el módulo de gráficos de la SDL
90    if (SDL_Init(SDL_INIT_VIDEO)<0) {
91      cerr « "No se puede iniciar SDL ("«SDL_GetError()«")"«endl;
92      exit(-1);
93    }

94    // Ponemos el modo de video:
95    // Dimensiones de la ventana: ancho x alto
96    // Profundidad de color: número de bits para representar el color de un pixel
97    pantalla=SDL_SetVideoMode(800,600,32,SDL_SWSURFACE|SDL_ANYFORMAT);
98    if (pantalla==0) {
99      cerr « "No se puede iniciar modo grafico ("«SDL_GetError()«")"«endl;
100     SDL_Quit();
101     exit(-1);
102   }

103   // Ponemos el título de la ventana
104   SDL_WM_SetCaption("Mi ventana",0);

105   // Bloqueamos la pantalla si es necesario
106   if (SDL_MUSTLOCK(pantalla)) {
107     if (SDL_LockSurface(pantalla)<0) {
108       cerr « "Error al bloquear la pantalla ("«SDL_GetError()«")"«endl;
109       SDL_Quit();
110       exit(-1);
111     }
112   }

113   // Dibujamos algo
114   Dibujar(pantalla);

115   // Desbloqueamos la pantalla si es necesario
116   if (SDL_MUSTLOCK(pantalla)) {
117     SDL_UnlockSurface(pantalla);
118   }

119   // Una vez que hemos dibujado en nuestra "pantalla virtual" (técnicamente se
120   // llama framebuffer) hemos de decirle a la SDL que actualice la información
121   // que aparece en el monitor en función de lo que hemos dibujado en el buffer
122   // Con la siguiente instrucción indicamos el rectangulo que deseamos
123   // actualizar en el monitor. Solo tiene sentido actualizar aquella zona del
124   // monitor en la que hallamos dibujado algo
```

```
125     SDL_UpdateRect(pantalla,0,0,512,512);

126     // Procesamos los eventos:
127     //   Pulsaciones de teclas
128     //   Movimientos de ratón
129     //   Acciones sobre la ventana (minimizar, cerrar, etc)
130     //   etc
131     ProcesaEventos(pantalla);

132     // Finalizamos
133     SDL_Quit();
134 }
```

9. Mas información

En la página oficial de SDL tienes muchos enlaces a tutoriales, programas que hacen uso de SDL y sobre todo a otras bibliotecas que amplían la funcionalidad de SDL para construir interfaces gráficos de usuario (GUI), dibujar formas complejas, etc. A continuación tienes algunos seleccionados:

Libros sobre SDL:

- *Alberto García Serrano*. PROGRAMACIÓN DE VIDEOJUEGOS CON SDL. Ediversitas Multimedia. 2003
- *Ernest Pazera, André LaMothe*. FOCUS ON SDL. Premier Press. 2003.
- *Loki Software Inc. width John R*. PROGRAMMING LINUX GAMES. Linux Journal Press. 2001.
- *Ron Penton, André LaMothe*. DATA STRUCTURES FOR GAME PROGRAMMERS. Premier Press. 2003.

Enlaces sobre SDL:

- `www.libsdl.org`. Página web oficial de la biblioteca SDL.
- `sdl-draw.sourceforge.net`. Biblioteca desarrollada sobre SDL que permite dibujar líneas, círculos, etc.
- `www.ferzkopp.net/~aschiffler/Software/SDL_gfx-2.0`. Idem que el anterior.
- `www.paragui.org`. Biblioteca desarrollada sobre SDL para el desarrollo de interfaces gráficos de usuario (GUI).

Enlaces sobre OpenGL:

SDL se puede integrar fácilmente con OpenGL para el tratamiento de gráficos 3D. Aquí tienes algunos enlaces sobre OpenGL:

- `www.opengl.org`. Página oficial de OpenGL.
- `www.mesa3d.org`. Página de la biblioteca Mesa 3D. Esta biblioteca tiene la misma API (interfaz de comandos) que OpenGL (es decir, que funciona exactamente igual) con la única diferencia de que es de libre distribución.
- `www.opengltutorial.co.uk`. Tutorial sobre OpenGL.
- `www.nigels.com/glt/glui`. Biblioteca construida sobre OpenGL para el desarrollo de interfaces gráficos de usuario.

Índice alfabético