PROGRAMACIÓN DE VIDEOJUEGOS CON SDL

Programación de Videojuegos con SDL

Alberto García Serrano

Índice.

Introducción *4*

Historia de los videojuegos *6*

La llegada del ordenador personal **6**

Sistemas Operativos o la torre de Babel **8**

Anatomía de un Videojuego *9*

Entrada. 9
Visualización. 9
Sonido. 10
Comunicaciones. 10
Game loop. 10

Los cimientos de un videojuego **11**
La idea. 12
Los elementos del juego 12
Manos a la obra 14

Primeros pasos con SDL *23*
Trabajando con SDL 24

Video **26**
Cargando y mostrando gráficos 28
Efectos especiales: Transparencias y alpha-blending 32
Otras funciones de interés 36

Gestión de eventos **38**
Tipos de eventos 39
Lectura de eventos 39
Eventos del teclado 40
Eventos de ratón 44
Eventos del joystick 45
Otros eventos 45

Joystick **47**
Recopilando información sobre el joystick 47
Leyendo el joystick 48

Audio **51**

CD-ROM **56**

El Window Manager **60**

Timming **60**

Librerías auxiliares para SDL *62*

SDL_ttf **62**

SDL_image **66**

SDL_mixer **69**
Sonidos 70
Música 73

Sprites: héroes y villanos *76*

Control de sprites **77**

Implementando los sprites **79**

Utilizando nuestra librería **86**

Un Universo dentro de tu ordenador *91*

Almacenando y mostrando tiles **93**

Diseñando el mapa **98**

Scrolling **100**

Enemigos, disparos y explosiones *105*

Tipos de inteligencia **105**

Comportamientos y máquinas de estado **107**

Disparos y explosiones **108**

¡Que comience el juego! ***117***

Enemigos **117**

Niveles **119**

Temporización **120**

Pantalla inicial, puntación y vidas **121**

¿Y ahora que? **136**

Instalación de SDL ***137***

Windows (VC++) **137**

Linux **141**

De C a C++ ***143***

Clases y objetos **144**

Herencia **147**

Polimorfismo **147**

Punteros y memoria **148**

Recursos ***150***

Bibliografía **150**
Programación 150
Programación de videojuegos 150
Gráficos 151

Enlaces **151**
Programación 151
Programación de videojuegos 151
Gráficos 152

Introducción

La imaginación es el motor de la creatividad, y la creatividad el motor de la inteligencia.

Muchas son las personas que opinan que la programación de ordenadores es una ciencia, otros lo ven como una ingeniería, sin embargo, algunos preferimos verla como un arte. Si tienes este libro entre tus manos, es porque te interesa la programación y también los videojuegos. Si es así, enhorabuena, porque estás a punto de entrar en un mundo realmente fascinante. La programación de videojuegos es muy diferente a la programación tradicional, requiere de una gran creatividad, pero también de una gran curiosidad y ganas de descubrir y, por que no, inventar cosas nuevas. De hecho, cuando creamos un videojuego, estamos creando un mundo nuevo, con su propio aspecto y sus propias leyes. En ese sentido el programador es como un dios dentro de ese mundo. La mayoría de las personas que se dedican a la programación se han planteado en algún momento el hacer un juego, y sin embargo son pocos los programadores que se deciden a hacerlo. ¿Hace falta algún don especial para hacer videojuegos? Probablemente no, pero hay muy buenos escritores de novelas que no son capaces de escribir poesía, y viceversa. En programación parece ocurrir lo mismo, hay programadores especialmente dotados para la programación de gestión y otros se desenvuelven mejor con una programación más cercana a la máquina.

Supondré que el lector ya posee conocimientos de programación, y más concretamente del lenguaje C. En el transcurso del libro, vamos a introducir las técnicas básicas utilizadas en la creación de videojuegos de forma práctica y a través de ejemplos, incluyendo el desarrollo de un juego completo. Vamos a utilizar para ello una librería multiplataforma llamada SDL (Simple Directmedia Layer) que nos va a permitir que nuestros juegos sean portables entre Windows, Linux, Macintosh, BeOs y otros. También vamos a introducir ciertas características de la programación orientada a objetos (POO) que nos ofrece C++, pero no debe preocuparse el lector desconocedor de la POO, porque dedicamos un apéndice completo para familiarizarnos con los elementos que vamos a usar de dicho lenguaje. La programación orientada a objetos es hoy necesaria a la hora de conseguir juegos de calidad profesional, por ello, introduciremos ciertos aspectos de este paradigma de programación.

Respecto a al grafismo utilizado en el juego de ejemplo que desarrollaremos durante el libro, he utilizado la librería de sprites GPL de Ari Feldman (http://www.arifeldman.com/), al cual agradezco, y animo a seguir ampliando.

Antes de acabar, quiero hacer una aclaración. Este no es un libro sobre programación y uso de SDL. Es un libro sobre programación de videojuegos, por lo tanto, no esperes una exhaustiva referencia sobre SDL. Iremos descubriendo los diferentes aspectos de esta librería según los necesitemos en nuestro camino a la creación de un videojuego, es por ello que habrá elementos de la librería que no formarán parte de los temas cubiertos por este libro.

Quiero, finalmente, dejar patente el objetivo que siempre he perseguido mientras escribía el presente texto: Mantener las cosas simples, sin entrar en tecnicismos innecesarios que distraigan al programador menos formal del objetivo final, que es dotarle de las herramientas necesarias para expresar toda su creatividad en forma de videojuego. No me cabe duda que hay cientos o miles de potenciales buenos programadores de juegos que, aún teniendo los conocimientos de programación necesarios, no han podido acceder a una información clara y carente de formalismos más

propios de un texto universitario. Espero que este sea un libro “para todos”, aunque ello signifique sacrificar en precisión y formalidad. El intentar trasladar la información contenida en el libro a un lenguaje sencillo y accesible me ha resultado a veces ciertamente una difícil tarea, por lo que pido excusas si en algún momento no he conseguido expresar las ideas con la suficiente simplicidad y claridad.

Puede contactar con el autor en la dirección de correo electrónico sdlintro@agserrano.com.

Capítulo 1

Historia de los videojuegos

Allá donde el tiempo pierde su memoria, grandes hombres y mujeres construyeron nuestro futuro.

La prehistoria de los videojuegos, se remonta a 1958. Bill Nighinbotthan presenta en una feria científica un aparato que permite, mediante unos potenciómetros, mover una pequeña raqueta en un tubo de rayos catódicos. Bill no fue capaz de intuir el basto potencial que aquel pequeño aparato tenía. Basándose en este aparato, Nolan Bushnell crea en 1972 un videojuego llamado *Pong*. Nollan, que sí es capaz de ver las posibilidades de este nuevo mercado, funda Atari, pero antes, Nollan ya había comercializado su primer videojuego: *Computer Space*. Años más tarde, en 1976, un empleado de Atari llamado Steve Jobs, ayudado por Steve Wozniak, crean un videojuego llamado *BreakOut*. Dos años después, ambos dejarán Atari para crear Apple Computer y pasar a la historia de la informática.

Steve Wozniak y Steve Jobs en su garaje, california. 1976

Es en 1978 cuando Taito lanza al mercado el famoso *Space Invaders*. Este juego era capaz de almacenar las puntuaciones máximas, convirtiéndose en todo un clásico.

En los siguientes años, comienzan a aparecer en el mercado nuevos videojuegos de excelente calidad, y que se convertirán en clásicos. Juegos como *Donkey Kong*, *Frogger*, *Galaga*, P*ac Man*, etc...

La llegada del ordenador personal

En la década de los 80, los ordenadores personales comienzan a popularizarse, y ponen en manos de particulares la posibilidad, no sólo de jugar a videojuegos creados para estos ordenadores, sino que ahora cualquiera puede adentrarse en el mundo de la programación de

videojuegos. Estos pequeños ordenadores de 8 bits hicieron las delicias de miles de aficionados a la informática. En España se hizo muy popular el Sinclair Spectrum. Un pequeño ordenador muy económico con un procesador Z80 y una memoria de 48 kilobytes. Otros ordenadores similares fueron el Amstrad 464, MSX y Commodore 64.

Commodore 64

De esta época tenemos juegos tan recordados como *Manic Miner*, *Xevious*, *Jet Set Willy*, *Arkanoid* y otros. A finales de los 80, surgen grupos de programadores en el Norte de Europa que comienzan a crear pequeñas joyas de la programación que intentaban exprimir el potencial de estas pequeñas máquinas al máximo. Concretamente estos grupos se crean en torno a la máquina de Commodore, gracias a sus posibilidades gráficas y de sonido, algo más avanzadas que la de sus competidores de la época. Estos pequeños programas es lo que hoy se ha dado en llamar *Demos*. Son por tanto los creadores de la actual *demoscene*.

En la década siguiente, aparecen ordenadores más potentes. Son equipos con microprocesadores de 16 bits y tienen entre 512 kbytes y 1 Mb de memoria. Quizás el más popular de estos fue el Amiga 500, también de Commodore, aunque sin olvidar al Atari ST o al Acorn Archimedes. Estos ordenadores tenían unas capacidades gráficas muy superiores a sus antecesores. Podían utilizar 4096 colores simultáneos, y su capacidad de sonido también comenzaba a ser bastante espectacular. De esta época son juegos como *Defender of the Crown*, *Barbarian*, *Elvira*, *Lemings* y *Maniac Mansión*.

Paralelamente a esta evolución, otro ordenador iba ganando adeptos a marchas forzadas. Sus capacidades gráficas y de sonido no eran mejores que la de sus competidores, de hecho eran bastante inferiores, sin embargo, lograron hacerse poco a poco con el mercado de los ordenadores personales, hasta el punto de reinar casi en solitario (con permiso del Apple Macintosh). Evidentemente hablamos del PC. Afortunadamente, al convertirse en el sistema mayoritario, sus capacidades gráficas y de sonido comienzan a avanzar a pasos agigantados. Una fecha clave es abril de 1987, cuando IBM presenta en el mercado la tarjeta gráfica VGA. Dejando obsoletas a las legendarias EGA y CGA. A la VGA la seguiría la SVGA, y después toda una pléyade de tarjetas con cada vez más memoria y aceleración gráfica 3D, llegando hasta las actuales Vodoo, las Gforce de Nvidia y las Radeon de ATI, con unas capacidades gráficas fuera de lo común a un precio muy asequible. Como es normal, esta época está marcada por juegos como *Quake*, *Counter-Strike (half life)*, *Heretic*, *Warcraft* y *Hexen*.

Sistemas Operativos o la torre de Babel

El ordenador PC, comercializado por IBM en el año 1982, ha pasado por varias etapas de existencia. Aquellos primeros PCs iban equipados con el sistema operativo PC-DOS, que no era más que una versión de IBM del MS-DOS de Microsoft. En los tiempos del MS-DOS la programación de videojuegos era bastante compleja, hacían falta grandes conocimientos de lenguaje ensamblador y de la estructura interna de la máquina, así como de la tarjeta de video y de sonido. Toda la programación se hacía directamente sobre el hardware. Es fácil imaginar los inconvenientes, había que preparar el juego para que funcionara con muchos modelos de tarjeta gráfica y también para los diferentes modelos de tarjetas de sonido (Adlib, Gravis Ultrasound (GUS), Sound Blaster). Es decir, hacer un juego, por simple que éste fuera, requería un esfuerzo muy alto, sin contar con que no siempre toda la información necesaria para acceder al hardware estaba al alcance de todo el mundo.

La llegada de Windows tampoco mejora demasiado el panorama, ya que no permite un acceso directo al hardware y su rendimiento gráfico es pobre. El acceso a los recursos del sistema también varía de un Sistema Operativo a otro. Se hace imprescindible, pues, crear una interfaz común para el desarrollo de videojuegos que permita el acceso estandarizado y con una interfaz común a los recursos del sistema. Algunas de estas propuestas son OpenGL y DirectX (éste último sólo para sistemas Windows).

En el caso de OpenGL, se nos ofrece una interfaz multiplataforma, sin embargo, se limita a ofrecernos una interfaz de acceso a gráficos (especializada en 3D, aunque con posibilidades de 2D) y no nos permite el acceso a otros recursos del sistema (teclado, sonido, joystick, timmers, etc...).

DirectX, sin embargo, está compuesto por varias sub-interfaces con el SO y el hardware, como DirectDraw (Gráficos 2D), Direct3D (Gráficos 3D), DirectInput (Entrada y Salida), DirectPlay (Capacidades de comunicación en redes TCP/IP), DirectSound (Acceso a hardware de sonido) y algún que otro componente más. Como vemos, DirectX tiene gran cantidad de capacidades, pero presenta dos inconvenientes. El primero es que sólo es válido para sistemas Windows, y el segundo es que por la propia naturaleza de la programación en Windows, su uso es algo engorroso y no apto para programadores con poca experiencia con las APIs de Windows y más concretamente con la interfaz COM+.

Afortunadamente, la empresa Loki Games, ante la necesidad de hacer portables los juegos de Windows a Linux y a otros sistemas operativos, crean una librería multiplataforma llamada SDL (Simple Directmedia Layer), que nos ofrece acceso a los recursos de la máquina (gráficos, sonido, entrada/salida, timming, etc...) mediante una interfaz coherente e independiente del SO. SDL, además, nos permite utilizar la interfaz OpenGL para gráficos 3D.

Capítulo 2

Anatomía de un Videojuego

Somos nuestra memoria, somos ese quimérico museo de formas inconstantes, ese montón de espejos rotos.
Jorge Luis Borges

Cada vez que juegas a tu videojuego preferido, dentro del ordenador están ocurriendo muchas cosas. Se hace patente que un factor importante en un juego es que se mueva con soltura y a buena velocidad. Todos los elementos del juego parecen funcionar de forma independiente, como con vida propia y a la vez. Sin embargo esto es sólo una apariencia, ya que dentro del programa se van sucediendo las diferentes fases de ejecución de forma secuencial y ordenada. En este capitulo vamos a tratar de dar una visión general y sin entrar en detalles de implementación de la anatomía de un videojuego. Vamos a ver qué partes lo componen y como se relacionan.

Entrada.

Un videojuego necesita comunicarse con el jugador a través de un dispositivo de entrada, como un Joystick o un teclado. La programación del sistema de Entrada/Salida puede parecer a priori algo fácil de implementar, y de hecho lo es. Sin embargo, si no queremos ver como nuestro estupendo juego se hace poco "jugable" para el jugador hay que cuidar mucho este aspecto. Una mala respuesta del juego ante una pulsación del teclado puede hacer frustrante la experiencia de juego para el jugador. Otro factor importante es el soporte de una variedad de dispositivos de entrada. Algunos jugadores preferirán usar el teclado al ratón y viceversa. Si además queremos que nuestro juego tenga un toque profesional, hay que dar la posibilidad al jugador de que defina las teclas con las que quiere jugar, además de dar soporte a los diferentes tipos de teclados internacionales.

Visualización.

La misión de esta capa es la de convertir el estado interno del juego en una salida gráfica. Ya sea en 2D o 3D, la calidad gráfica es un factor importantísimo. Normalmente, el programador trabaja con unos gráficos creados por él mismo, y cuya calidad dependerá de las dotes artísticas de éste. En una siguiente fase, un artista gráfico crea y sustituye estas imágenes por las definitivas. Uno de los elementos principales de un buen juego es un buen motor gráfico, capaz de mover una gran cantidad de Sprites en un juego 2D o una gran cantidad de polígonos (triángulos) en el caso de un juego en 3D. En el capítulo 6 veremos qué es un Sprite.

En el diseño del motor gráfico, siempre hay que llegar a un compromiso entre la compatibilidad y el rendimiento. Hay tarjetas gráficas en el mercado con unas capacidades casi sobrenaturales, pero que no son estándar, sin embargo, tampoco podemos dar la espalda a los jugadores "profesionales" que equipan sus ordenadores con la última aceleradora 3D del mercado.

Sonido.

El sonido es un aspecto al que, a veces, no se le da la importancia que merece. Tan importante como unos buenos gráficos es un buen sonido. Tanto el sonido como la banda sonora de un juego son capaces de transmitir una gran cantidad de sensaciones al jugador, y eso es, al fin y al cabo, lo que se busca. Tanto como la calidad, la buena sincronización con lo que va sucediendo en pantalla es de suma importancia.

Comunicaciones.

Cada vez más, los juegos en red van ganando en adeptos. La comunicación se realiza normalmente sobre redes TCP/IP. Esto nos permite jugar contra otros jugadores situados en nuestra propia red local o en Internet. La comunicación entre máquinas se lleva a cabo mediante la programación de sockets. Afortunadamente SDL nos provee de herramientas muy potentes para este tipo de comunicación, aunque esto es algo que queda fuera del alcance de este libro.

Game loop.

El "game loop" o bucle de juego es el encargado de "dirigir" en cada momento que tarea se está realizando. En la figura 2.1. podemos ver un ejemplo de game loop. Es sólo un ejemplo, y aunque más o menos todos son similares, no tienen por que tener exactamente la misma estructura. Analicemos el ejemplo. Lo primero que hacemos es leer los dispositivos de entrada para ver si el jugador ha realizado alguna acción. Si hubo alguna acción por parte del jugador, el siguiente paso es procesarla, esto es, actualizar su posición, disparar, etc... dependiendo de que acción sea. En el siguiente paso realizamos la lógica de juego, es decir, todo aquello que forma parte de la acción del juego y que no queda bajo control del jugador, por ejemplo, el movimiento de los enemigos, cálculo de trayectoria de sus disparos, comprobamos colisiones entre la nave enemiga y la del jugador, etc... Fuera de la lógica del juego quedan otras tareas que realizamos en la siguiente fase, como son actualizar el scroll de fondo (si lo hubiera), activar sonidos (si fuera necesario), realizar trabajos de sincronización, etc.. Ya por último nos resta volcar todo a la pantalla y mostrar el siguiente frame. Esta fase es llamada "fase de render".

Normalmente, el game loop tendrá un aspecto similar a lo siguiente:

```
int done = 0;
while (!done) {
   // Leer entrada
   // Procesar entrada (si se pulsó ESC, done = 1)
   // Lógica de juego
   // Otras tareas
   // Mostrar frame
}
```

Antes de que entremos en el game loop, tendremos que hacer un Múltiples tareas, como inicializar el modo gráfico, inicializar todas las estructuras de datos, etc...

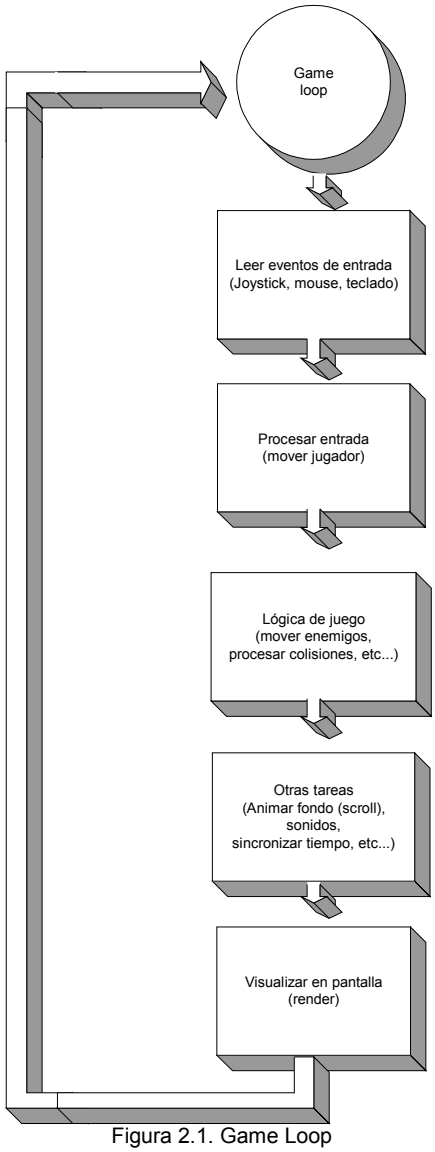


Figura 2.1. Game Loop

Los cimientos de un videojuego

Como toda buena obra de arte, un buen videojuego comienza con una buena idea, pero no basta con eso, hay que hacer que esa idea se transforme en una realidad. Es este capítulo vamos a desarrollar un pequeño juego que ilustre las fases y las ideas implicadas en el diseño. Desde el momento que el diseño de un videojuego no es una ciencia exacta, sino que tiene mucho de arte, este capítulo ha de tomarse como una guía general. Como aún no hemos visto nada sobre SDL ni sobre programación gráfica, vamos a crear un juego de aventuras tipo "conversacional". Los que llevéis más tiempo en esto de los ordenadores, conoceréis ya como funcionan estos juegos (algunos recordarán *"La aventura original"* y *"Don Quijote"* por nombrar dos de ellos). Básicamente, el ordenador nos describe de forma textutal los escenarios sobre los que nos movemos. Podemos dar ordenes a nuestro protagonista mediante frases que el ordenador interpreta. Por ejemplo, podemos dar una orden como: "coger amuleto". Algunos de estos juegos podían interpretar frases muy complejas, pero nosotros nos limitaremos a dos palabras con la forma *verbo nombre*. Por ejemplo: "ir norte" o "usar llave".

La idea.

Hoy en día no basta con unos impresionantes gráficos y un buen sonido para hacer un juego. Es muy importante tener una idea sobre la que se asiente el juego, o lo que es lo mismo, un buen argumento. Inventar una historia en la que pueda sumergirse el jugador y sentirse parte de ella es un trabajo más de un escritor que de un programador. Es por ello que en las empresas dedicadas a los videojuegos, hay personas dedicadas exclusivamente a escribir guiones e historias. Como supongo que eres un buen programador y como para ser buen programador hay que ser una persona creativa, estoy seguro que no te costara conseguir una buena idea. Si la musa no te inspira, siempre puedes hacer "remakes" de otros juegos (intenta poner algo de tu propia cosecha para mejorarlo), basarte en alguna película o algún libro (Poe puede haber inspirado los escenarios de algunos de los juegos más terroríficos), hacer algún juego con temática deportiva o incluso basarte en algún sueño o pesadilla que hayas tenido (y no me digas que *Doom* no parece sacado de la pesadilla de un programador adolescente).

Pasemos a la práctica. Vamos a crear un argumento para nuestro juego de ejemplo:

"Fernando es un chico de 17 años, su vida es aparentemente normal. Su vecina Carolina, con la que trata de salir desde los 11 años, tiene una especial atracción por todo lo paranormal. No lejos de sus casas hay una pequeña vivienda que según cuentan está embrujada y a la cual llaman la mansión del terror. Según Carolina, todo aquél que entra queda atrapado y sin salida posible. Fernando, que además de no creer en fantasma es muy dado a las bravuconadas, ha hecho una apuesta con Carolina. Si consigue salir de la casa, irá con él al baile de fin de curso. Fernando ha entrado por una ventana, que se ha cerrado de golpe cortando su única salida.
¿Conseguirá Fernando salir de la mansión del terror?
¿Conseguirá Fernando ir al baile con Carolina?
Todo depende de ti ahora..."

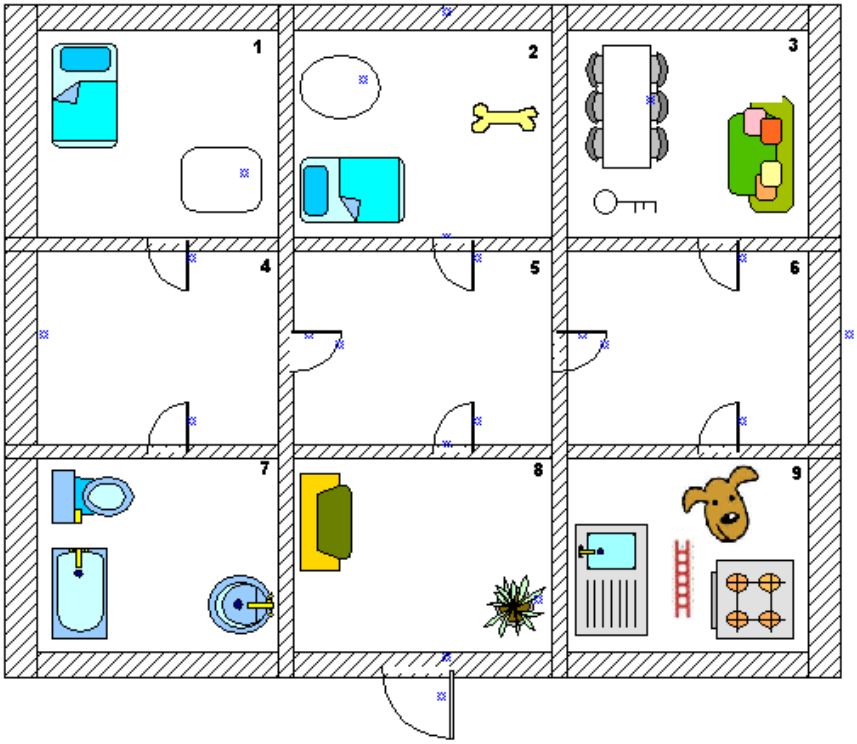
Como puedes ver, tampoco hay que ser un gran literato para crear una historia. Aun siendo una historia muy simple, se consigue que el jugador tenga una meta y luche por algo. Si tienes la suerte de saber dibujar, puedes incluso crear un *story board* con los detalles de la historia y los escenarios del juego.

Los elementos del juego

Los detalles son importantes y hay que tenerlos claros.

¿Cómo va a ser la mansión?
¿Cuántas habitaciones tiene?
¿Qué objetos podemos encontrar?
¿Cómo conseguimos ganar?
¿Cuáles son las dificultades?

Es necesario hacerse todo este tipo de preguntas para tener claro como va a desarrollarse una partida. Empecemos por nuestra misión. Para ganar, hemos de lograr salir de la casa. Para ello hemos de conseguir la llave que abre la puerta de la misma. Vamos a crear una mansión de pequeño tamaño, con sólo 9 habitaciones (utilizaremos un mapa de 3x3). Veamos como queda el mapa.



Hemos numerado las habitaciones de la casa para tener una referencia inequívoca de cada una. Una vez tenemos claro el mapa, situamos los objetos que nos permitirán cumplir nuestra misión:

Objeto 1 – Hueso – Está en la habitación 2.
Objeto 2 – Llave – Está en la habitación 3.
Objeto 3 – Escalera – Está en la habitación 9.
Objeto 4 – Perro – Está en la habitación 9.

y una lista de reglas sobre cómo usarlos:

Habitación 3 – Para coger la llave hay que tener la escalera, ya que la llave está sobre la lámpara del techo.

Habitación 9 – Para poder coger la escalera hay que darle un hueso al perro rabioso.

Habitación 8 – Para abrir la puerta y salir de la mansión hay que tener la llave.

Ya con el mapa finalizado, los objetos y su función dentro del juego, vamos a ver qué ordenes podemos darle a nuestro protagonista y sobre qué objetos.

Coger – Hueso, Llave, Escaleras.
Dar – Hueso.
Usar – Llave.
Ir

Es importante definir bien que acciones son válidas sobre cada objeto. Ahora que hemos pensado y plasmado por escrito los elementos del juego, es el momento de pasar a la acción empezar a escribir código.

Manos a la obra

Una de las claves a la hora de hacer un buen programa es partir de unas buenas estructuras de datos. Han de permitirnos cierta flexibilidad y a su vez no deben ser demasiado complejas de manejar. Empecemos por el mapa. La unidad básica más pequeña para poder describir el mapa es una habitación, así que vamos a crear el mapa en memoria como una serie de habitaciones. Para describir una habitación usaremos la siguiente estructura de datos:

```
// Estructura de datos para las
// habitaciónes
struct habitacion {
   char desc[255];
   int norte;
   int sur;
   int este;
   int oeste;
} habitaciones[10];
```

Hemos creado un array de 10 habitaciones (de 0 a 9. Aunque usaremos 9. Del 1 al 9), que son precisamente las habitaciones que tiene nuestro mapa. Si recuerdas, en el mapa pusimos una numeración a cada habitación. Esta numeración ha de coincidir con su índice en el array. Veamos el significado de cada campo.

El primero es la descripción textual de la habitación. Aquí describimos lo que el jugador puede ver. Esta descripción es siempre fija, por lo que omitiremos indicar si hay algún objeto en la habitación, ya que estos podrían cambiar de sitios o ser cogidos y ya no estar en la habitación.

Los siguientes cuatro campos son los de dirección, correspondientes a Norte, Sur, Este y Oeste, e indican a la habitación que llegaremos si optamos por dar la orden de movernos en una de éstas direcciones. Veámoslo con un ejemplo:

```
// Habitación 4
strcpy(habitaciones[4].desc,"Estas en el pasillo.");
habitaciones[4].norte=1;
habitaciones[4].sur=7;
habitaciones[4].este=5;
habitaciones[4].oeste=0;
```

El primer campo no necesita mucha explicación, simplemente muestra el texto descriptivo de donde se haya el jugador. Los cuatro siguientes nos indican que si el jugador decide ir al norte, llegará a la habitación 1, si decide ir al sur llegará a la habitación 7 y así sucesivamente. Si ponemos uno de estos campos a 0, significa que el jugardor no puede moverse en esa dirección. Por lo tanto, si en nuestro programa tenemos una variable llamada `habitacionActual` en la que se almacena la habitación en la que se encuantra el jugador en este preciso instante, para hacer el movimiento usaremos algo como:

```
habitacionActual=habitaciones[habitacionActual].norte;
```

Simple, ¿no? Sigamos con otra estructura de datos importante: la encargada de describir los objetos. Echémosle un vistazo:

```
// Estructuras de datos para los objetos
struct objeto {
   int estado;      // Estado del objeto
   char desc1[80];  // Descripción para estado 1
   char desc2[80];  // Descripción para estado 2
```

```
   int hab;          // Habitación en la que se
                     // encuentra
   int lotengo;      // Indica si tengo este objeto
                     // en mi inventario
} objetos[5];
```

Al igual que con las habitaciones, hemos creado un array de 5 elementos para los objetos (aunque usaremos 4. Del 1 al 4.). Veamos el significado de los campos de la estructura.

El campo `estado` es para objetos que pueden tener dos estados diferentes. Por ejemplo, imagina una linterna, que podría estar encendida o apagada. Los campos `desc1` y `desc2`, contienen la descripción del objeto cuando está en estado 1 y cuando está en estado 2. En `desc1` habría algo como "`una linterna apagada`" y en `desc2` tendríamos "`una linterna encendida`". Puede que un objeto sólo tenga un estado, por ejemplo, una llave. En este caso el campo `desc2` puede dejarse vació o contener el mismo texto que `desc1`, ya que nunca será mostrado.

El campo `hab`, indica en que habitación se encuentra el objeto, y el campo `lotengo` si está a 0 significa que el jugador no posee este objeto y si está a 1 es que lo tiene en su inventario. El siguiente ejemplo inicializa un objeto del juego:

```
// perro
objetos[PERRO].estado=1;
strcpy(objetos[PERRO].desc1,"un perro rabioso");
strcpy(objetos[PERRO].desc2,"un perro comiéndose un hueso");
objetos[PERRO].hab=9;
objetos[PERRO].lotengo=0;
```

Este código inicializa el objeto perro. Por claridad hemos usado la constante PERRO en lugar de usar directamente su posición en el array. En el caso del perro, una vez que le damos el hueso, pasa al estado 2, en el que ya no es peligroso.

Ahora que ya tenemos definidas las estructuras de datos, vemos como quedaría el *game loop* de nuestro juego.

```
// Inicialización de estados
while (!done) {

   // Mostramos información de la habitación.

   // Leemos la entrada del jugador

   // Procesamos la entrada del jugador
}
```

En la inicialización de estados establecemos los valores iniciales de las variables que van a definir el estado del juego. Por ejemplo, la variable `habitacionActual`, indica en qué habitación nos encontramos. Tenemos que darle un valor inicial, que es la habitación donde comienza el jugador la partida. En este caso, la habitación 1.

Ya dentro del *game loop*, la primera tarea es mostrar la descripción del lugar donde se haya el jugador. Para este juego, conviene poner esta fase al principio del bucle, sin embargo, en otros (como en el caso de *1945*, que desarrollaremos en los siguientes capítulos) nos convendrá más situarlo en la parte final del mismo.

Hay tres informaciones que necesita el jugador: la descripción de la habitación, si se encuentra algún objeto en la misma y por último, cuáles son las salidas válidas de la estancia.

Empecemos por el principio. El siguiente código es el encargado de mostrar la descripción de la habitación:

```
// Descripción
printf("\n\n%s",habitaciones[habitacionActual].desc);
```

Como ves, no tiene nada de complicado. Simplemente escribe por pantalla la descripción de la habitación contenida en la estructura de datos que vimos antes. El siguiente paso es mostrar los objetos que hay en la habitación, si es que hubiera alguno.

```
// Mostramos si hay algún objeto
for (i=1 ; i<=nobjetos ; i++) {
  if (objetos[i].hab == habitacionActual) {
    printf("\nTambien puedes ver ");
    // mostramos la descripción del objeto según
    // su estado
    if (objetos[i].estado == 1) {
      printf("%s",objetos[i].desc1);
      } else {
      printf("%s",objetos[i].desc2);
    }
  }
}
```

Con el bucle `for` comprobamos cada uno de los objetos. Si está en la habitación en la que se encuentra el jugador en ese preciso instante, muestra la descripción del objeto según su estado. Ya sólo nos resta mostrar las direcciones posibles que el jugador puede tomar.

```
// Mostramos las posibles salidas
printf("\nPuedes ir dirección ");
if (habitaciones[habitacionActual].norte != 0)
  printf ("Norte ");
if (habitaciones[habitacionActual].sur != 0)
  printf ("Sur ");
if (habitaciones[habitacionActual].este != 0)
  printf ("Este ");
if (habitaciones[habitacionActual].oeste != 0)
  printf ("Oeste ");
```

Si el campo de dirección de la estructura de datos de la habitación contiene un valor distinto a 0, significa que ese camino lleva a otra habitación distinta y, por lo tanto, es una salida válida.

Lo siguiente que se hace en el *game loop*, es leer la entrada del jugador. Como habíamos dicho antes, y con el fin de simplificar, la entrada del usuario constará de dos palabras: un verbo y un objeto sobre el que realizar la acción.

```
// Leemos la entrada del jugador
printf("\n>>> ");
scanf("%s%s",verbo,nombre);
```

`scanf` leerá dos palabras separadas por un espacio, que es justamente lo que necesitamos.

Por último, queda por procesar la entrada del jugador.

```
// coger
if (!strcmp(verbo,"coger")) {
   accion = 0;

   // Hueso
   if (!strcmp(nombre,"hueso") && objetos[HUESO].hab == habitacionActual) {
      accion = 1;
      objetos[HUESO].hab=0;
      objetos[HUESO].lotengo=1;
      printf("\nHas cogido el hueso.");
   }
}
```

Este fragmento de código muestra como se procesa la entrada del jugador, en concreto para el verbo coger y el objeto hueso. Antes de realizar cualquier acción, debemos asegurarnos de que el objeto está en la habitación en la que se encuentra el jugador. Si todo es correcto, actualizamos la estructura de datos del objeto hueso e informamos al jugador de que se realizó la acción.

La variable `accion` es lo que se llama una variable tipo bandera o flag. Su misión es informar que se ha realizado una acción o no, en un punto más adelantado del código. Necesitamos saber si se realizó alguna acción para informar covenientemente al jugador en caso de no poder llevar a cabo su orden. Si hemos cogido un objeto, ya no está en ninguna habitación, esto lo indicamos poniendo a 0 el campo `hab` de la estructura de datos del objeto. El código completo del juego queda así.

```
// La mansión del terror
// Programación de videojuegos con SDL
// (c) Alberto García Serrano


#include <stdio.h>
#include <string.h>

/***
   Declaración de estructuras de datos
 ***/

#define HUESO 1
#define LLAVE 2
#define ESCALERA 3
#define PERRO 4

// Estructuras de datos para los objetos
struct objeto {
   int estado;        // Estado del objeto
   char desc1[80];    // Descripción para estado 1
   char desc2[80];    // Descripción para estado 2
   int hab;        // Habitación en la que se encuentra
   int lotengo;    // Indica si tengo este objeto en mi inventario
} objetos[5];


// Estructura de datos para las habitaciónes
struct habitacion {
   char desc[255];
   int norte;
   int sur;
   int este;
   int oeste;
} habitaciones[10];


void main() {

   /***
      Inicialización de estructuras de datos
    ***/

   // Inicialización de objetos

   // hueso
   objetos[HUESO].estado=1;
   strcpy(objetos[HUESO].desc1,"un hueso");
   strcpy(objetos[HUESO].desc2,"un hueso");
   objetos[HUESO].hab=2;
   objetos[HUESO].lotengo=0;

   // llave
   objetos[LLAVE].estado=1;
   strcpy(objetos[LLAVE].desc1,"una llave sobre la lampara");
   strcpy(objetos[LLAVE].desc2,"una llave sobre la lampara");
   objetos[LLAVE].hab=3;
   objetos[LLAVE].lotengo=0;

   // escalera
   objetos[ESCALERA].estado=1;
   strcpy(objetos[ESCALERA].desc1,"una escalera");
   strcpy(objetos[ESCALERA].desc2,"una escalera");
   objetos[ESCALERA].hab=9;
   objetos[ESCALERA].lotengo=0;

   // perro
   objetos[PERRO].estado=1;
   strcpy(objetos[PERRO].desc1,"un perro rabioso");
   strcpy(objetos[PERRO].desc2,"un perro comiendose un hueso");
   objetos[PERRO].hab=9;
   objetos[PERRO].lotengo=0;



   // Datos del mapa
```

```
   // Habitación 1
   strcpy(habitaciones[1].desc,"Estas en una pequeña habitación pintada de blanco. Junto a ti
puedes ver una cama y una mesita de noche.");
   habitaciones[1].norte=0;
   habitaciones[1].sur=4;
   habitaciones[1].este=0;
   habitaciones[1].oeste=0;

   // Habitación 2
   strcpy(habitaciones[2].desc,"Estas en una habitación pintada de verde. Junto a ti puedes ver
una cama y una mesita de noche.");
   habitaciones[2].norte=0;
   habitaciones[2].sur=5;
   habitaciones[2].este=0;
   habitaciones[2].oeste=0;

   // Habitación 3
   strcpy(habitaciones[3].desc,"Estas en el salón de la casa. Puedes ver una gran mesa rodeada
de sillas.");
   habitaciones[3].norte=0;
   habitaciones[3].sur=6;
   habitaciones[3].este=0;
   habitaciones[3].oeste=0;

   // Habitación 4
   strcpy(habitaciones[4].desc,"Estas en el pasillo.");
   habitaciones[4].norte=1;
   habitaciones[4].sur=7;
   habitaciones[4].este=5;
   habitaciones[4].oeste=0;

   // Habitación 5
   strcpy(habitaciones[5].desc,"Estas en el pasillo.");
   habitaciones[5].norte=2;
   habitaciones[5].sur=8;
   habitaciones[5].este=6;
   habitaciones[5].oeste=4;

   // Habitación 6
   strcpy(habitaciones[6].desc,"Estas en el pasillo.");
   habitaciones[6].norte=3;
   habitaciones[6].sur=9;
   habitaciones[6].este=0;
   habitaciones[6].oeste=5;

   // Habitación 7
   strcpy(habitaciones[7].desc,"Estas en el típico servicio, con sus típicas piezas.");
   habitaciones[7].norte=4;
   habitaciones[7].sur=0;
   habitaciones[7].este=0;
   habitaciones[7].oeste=0;

   // Habitación 8
   strcpy(habitaciones[8].desc,"Estas en la entrada de la casa. Puedes ver la puerta cerrada.");
   habitaciones[8].norte=5;
   habitaciones[8].sur=0;
   habitaciones[8].este=0;
   habitaciones[8].oeste=0;

   // Habitación 9
   strcpy(habitaciones[9].desc,"Estas en la cocina.");
   habitaciones[9].norte=6;
   habitaciones[9].sur=0;
   habitaciones[9].este=0;
   habitaciones[9].oeste=0;


   /***
      Inicialización del estado de juego.
    ***/

   // variable que indica la habitación en la que estamos
   int habitacionActual = 1;

   // variable que indica cuantos objetos hay
   int nobjetos = 4;
```

```
/***
   game loop
 ***/

char verbo[30], nombre[30];
int i, accion;
int done = 0;

while (!done) {

   // Mostramos información de la habitación.

   // Descripción
   printf("\n\n%s",habitaciones[habitacionActual].desc);

   // Mostramos si hay algun objeto
   for (i=1 ; i<=nobjetos ; i++) {
      if (objetos[i].hab == habitacionActual) {
         printf("\nTambien puedes ver ");
         // mostramos la descripción del objeto según su estado
         if (objetos[i].estado == 1) {
            printf("%s",objetos[i].desc1);
         } else {
            printf("%s",objetos[i].desc2);
         }
      }
   }

   // Mostramos las posibles salidas
   printf("\nPuedes ir dirección ");
   if (habitaciones[habitacionActual].norte != 0)
      printf ("Norte ");
   if (habitaciones[habitacionActual].sur != 0)
      printf ("Sur ");
   if (habitaciones[habitacionActual].este != 0)
      printf ("Este ");
   if (habitaciones[habitacionActual].oeste != 0)
      printf ("Oeste ");


   // Leemos la entrada del jugador
   printf("\n>>> ");
   scanf("%s%s",verbo,nombre);



   // Procesamos la entrada del jugador

   // coger
   if (!strcmp(verbo,"coger")) {
      accion = 0;

      // Hueso
      if (!strcmp(nombre,"hueso") && objetos[HUESO].hab == habitacionActual) {
         accion = 1;
         objetos[HUESO].hab=0;
         objetos[HUESO].lotengo=1;
         printf("\nHas cogido el hueso.");
      }

      // Llave
      if (!strcmp(nombre,"llave") && objetos[LLAVE].hab == habitacionActual) {
         accion = 1;
         // para coger la llave necesitamos la escalera
         if (objetos[ESCALERA].lotengo == 1) {
            objetos[LLAVE].hab=0;
            objetos[LLAVE].lotengo=1;
            printf("\nTras subirte a la escalera, alcanzas la llave.");
         } else {
            printf("\nNo alcanzo la llave. Está demasiado alta.");
         }
      }

      // Escaleras
      if (!strcmp(nombre,"escalera") && objetos[ESCALERA].hab == habitacionActual) {
         accion = 1;
         if (objetos[PERRO].estado == 2) {
```

```
               objetos[ESCALERA].hab=0;
               objetos[ESCALERA].lotengo=1;
               printf("\nHas cogido las escaleras.");
            } else {
               printf("\nEl perro gruñe y ladra y no te deja cogerlo.");
            }
         }


         if (accion == 0)
            printf("\nNo puedes hacer eso.");

      }


      // dar
      if (!strcmp(verbo,"dar")) {
         accion = 0;

         // Hueso
         if (!strcmp(nombre,"hueso") && objetos[HUESO].lotengo == 1 && objetos[PERRO].hab ==
habitacionActual) {
            accion = 1;
            objetos[HUESO].lotengo=0;
            objetos[PERRO].estado=2;
            printf("\nEl perro coge el hueso y se retira a comerselo tranquilamente.");
         }

         if (accion == 0)
            printf("\nNo puedes hacer eso.");
      }


      // usar
      if (!strcmp(verbo,"usar")) {
         accion = 0;

         // Hueso
         if (!strcmp(nombre,"llave") && objetos[LLAVE].lotengo == 1 && habitacionActual == 8) {
            accion = 1;
            printf("\nENHORABUENA!!! Has escapado de la mansión del terror.");
            done = 1;
         }

         if (accion == 0)
            printf("\nNo puedes hacer eso.");
      }


      // ir
      if (!strcmp(verbo,"ir")) {
         accion = 0;

         // norte
         if (!strcmp(nombre,"norte") && habitaciones[habitacionActual].norte != 0) {
            accion = 1;
            habitacionActual=habitaciones[habitacionActual].norte;
         }

         // sur
         if (!strcmp(nombre,"sur") && habitaciones[habitacionActual].sur != 0) {
            accion = 1;
            habitacionActual=habitaciones[habitacionActual].sur;
         }

         // este
         if (!strcmp(nombre,"este") && habitaciones[habitacionActual].este != 0) {
            accion = 1;
            habitacionActual=habitaciones[habitacionActual].este;
         }

         // oeste
         if (!strcmp(nombre,"oeste") && habitaciones[habitacionActual].oeste != 0) {
            accion = 1;
            habitacionActual=habitaciones[habitacionActual].oeste;
         }
```

```
         if (accion == 0)
            printf("\nNo puedes hacer eso.");
      }

   }
}
```

Capítulo

3

Primeros pasos con SDL

Iniciar el camino es fácil, finalizarlo también. Lo meritorio es recorrerlo.

Tanto si utilizas Windows, Linux o cualquier otro sistema operativo, SDL te va a ahorrar un montón de trabajo y de dolores de cabeza. El apéndice A está dedicado a la instalación de la librería para ambos sistemas operativos, así que si aún no lo tienes instalado es un buen momento para saltar a dicho apéndice e instalarlo. En este capítulo, vamos a introducirnos en el manejo de esta librería. No describiremos todas y cada una de las partes que la componen, ya que para ello necesitaríamos un libro dedicado a ello en exclusiva. Veremos, sin embargo, lo más importante y lo necesario para poder construir un videojuego.

SDL es el acrónimo de Simple DirectMedia Layer. Al igual que DirectX en Windows, SDL nos ofrece una completa API para el desarrollo de juegos, demos y todo tipo de aplicaciones multimedia con independencia del sistema operativo y la plataforma. Actualmente SDL soporta Linux, Windows, BeOS, MacOS, MacOS X, FreeBSD, OpenBSD, BSD/OS, Solaris, y IRIX. Aunque tiene portes no oficiales a Windows CE, AmigaOS, Atari, QNX, NetBSD, AIX, OSF/Tru64, y SymbianOS.

SDL está compuesto por subsistemas separados que nos ofrecen soporte a diferentes partes del hardware.

Subsistema de Video.

Es nuestra interfaz con el hardware de video. Nos permite operaciones básicas como la inicialización de modos gráficos, trabajo con colores y paletas de color, manejo de *surfaces,* colores transparentes, alpha blending y otras (Veremos que significan estos términos cuando profundicemos en el subsistema de video).

Subsistema de Audio.

Es el subsistema encargado de todo lo que tenga que ver con la reproducción de sonidos. Nos va a permitir reproducir archivos .wav de forma relativamente sencilla.

Subsistema de manejo de eventos.

Los eventos son la base de la interactividad. Los necesitamos para saber si el jugador quiere mover la nave a la izquierda o por el contrario quiere hacer un disparo. Básicamente nos va a permitir conocer el estado del teclado y del ratón en cualquier momento.

Joysticks.

Seguramente te estarás preguntando por qué no se controla el joystick desde el subsistema de eventos. Yo también me lo pregunto y ciertamente no encuentro una buena razón para ello. Probablemente la razón sea que existen multitud de sistemas diferentes de joysticks en el mercado, quizás los suficientes para justificar un módulo separado. En cualquier caso, gracias a este subsistemas, nuestros juegos podrán ser controlados mediante un joystick.

CD-ROM.

Efectivamente, SDL nos permite el acceso a todas las funciones del CD-ROM. Las posibilidades son muy atractivas. Por ejemplo, si distribuyes tu juego en CD, éste puede tener una pista sonora y reproducirla en background mientras el jugador juega. Interesante ¿no?

Threads.

Sin duda sabes lo que es la multitarea. Puedes ejecutar múltiples programas al mismo tiempo. La idea de thread (hilo en español) es similar. Son pequeños procesos que se lanzan simultáneamente, pero que pertenecen a un mismo programa padre. Por ejemplo, un juego puede tener un thread encargado de reproducir una música y otro que va calculando la posición de los enemigos. Cada sistema operativo nos ofrece su propia forma de trabajar con los threads, pero SDL se jacta de ofrecer una API multiplataforma para manejo de threads.

Timers.

Es posible que alguna vez hayas probado alguno de esos juegos antiguos (muy antiguos) de cuando los PC corrían a 8 ó 16Mhz en tu flamante Pentium IV. Si el juego no había tenido en cuenta la velocidad del ordenador, probablemente lo verás funcionar tan rápido que serás incapaz de jugar. Si el juego hubiera estado bien diseñado, habría tenido en cuenta esta eventualidad, ya que no todos los ordenadores funcionan a la misma velocidad. SDL nos ofrece acceso a timers de alta resolución (o al menos de una resolución aceptable) para hacer que nuestro juego trabaje a la misma velocidad con independencia de la máquina en la que se ejecute.

Extensiones.

Además de estos subsistemas básicos, SDL tiene ciertas extensiones que nos harán la vida más fácil, como son SDL_image que permite trabajar con múltiples formatos de archivos gráficos, SDL_ttf para manejo de fuentes true type, SDL_net para acceso a redes TCP/IP y SDL_mixer que mejora sustancialmente el subsistema de audio de SDL.

Trabajando con SDL

Es hora de ponerse manos a la obra y poner a trabajar al compilador.

Antes de llamar a cualquier función de SDL, hay que inicializarla y para ello se utiliza la función `SDL_Init(Uint32 flags)`. Veámoslo con un ejemplo.

```
#include <stdlib.h>
#include "SDL.h"

main(){
    if (SDL_Init(SDL_INIT_AUDIO|SDL_INIT_VIDEO)< 0) {
        printf("No se puede iniciar SDL: %s\n",SDL_GetError());
```

```
        exit(1);
    }
    atexit(SDL_Quit);
}
```

En este fragmento de código hay varias cosas interesantes. Empecemos por la función SDL_Init. Como parámetro acepta un argumento de tipo `Uint32`. Como te dije antes, SDL es multiplataforma, por lo que tiene definidos una serie de tipos de datos que serán iguales con independencia del sistema en el que corra. La U quiere decir que el tipo de datos es sin signo (unsigned), int, como podrás adivinar es por que es un entero y 32, porque es un entero de 32 bits. Si en vez de U, usamos la S (signed) estaremos hablando de un entero con signo. Para el número de bits, los posibles valores son 8,16,32 o 64. Así, por ejemplo, `Uint16`, hará referencia a un entero de 16 bits. Veamos en detalle los parámetros que hemos pasado a `SDL_Init()`. `SDL_INIT_AUDIO|SDL_INIT_VIDEO`. Con este parámetro le decimos a SDL que sólo queremos inicializar el subsistema de audio y de video. Los posibles valores son:

SDL_INIT_VIDEO
SDL_INIT_AUDIO
SDL_INIT_TIMER
SDL_INIT_CDROM
SDL_INIT_JOYSTICK
SDL_INIT_EVERYTHING

Los parámetros se pasan separados por la barra vertical (|). Si quisiéramos inicializar además del audio y el video el cd-rom, los parámetros serían los siguientes:

```
SDL_INIT_AUDIO|SDL_INIT_VIDEO|SDL_INIT_CDROM
```

Si lo queremos activar todo, pasaremos como único parámetro `SDL_INIT_EVERYTHING`. Una vez inicializado SDL, si necesitamos inicializar otro subsistema podemos hacerlo con la función SDL_InitSubSystem(Uint32 flags) de la siguiente manera.

```
// Inicializamos el CD-ROM
if (SDL_InitSubSystem(SDL_INIT_CDROM) == -1) {
  printf("No se puede iniciar el cdrom: %s\n",SDL_GetError());
  exit(1);
}
```

Como habrás adivinado, la función SDL_GetError() devuelve el último error interno de SDL en formato cadena. Cuando una aplicación SDL es inicializada, se crean dos archivos, *stderr.txt* y *stdout.txt*. Durante la ejecución del programa, cualquier información que se escriba en la salida de error estándar se escribe en el archivo *stderr.txt*. Igualmente, cualquier información que se escriba en la salida estándar se guardará en el archivo *stdout.txt*. Esto nos va a ser de gran ayuda a la hora de depurar programas. Una vez finalizada la ejecución, si estos archivos están vacíos son borrados automáticamente, si no, estos permanecen intactos.

Al igual que inicializamos SDL, cuando hemos terminado el trabajo hemos de cerrarlo. La función encargada de esta tarea es SDL_Quit(). En nuestro programa de ejemplo hemos usado la siguiente la línea `atexit(SDL_Quit)`. La función atexit() toma como parámetro a otra función a la que llama justo antes de que finalice la ejecución del programa. En nuestro caso, antes de que finelice nuestro programa (ya sea por un error o porque el usuario forzó la salida) se llamará a la función SDL_Quit(). Una de las particularidades de `atexit()` es que la función a la que debe llamar no tiene que tener parámetros ni devolver nada.

En el caso de que queramos finalizar sólo un subsistema de SDL usaremos la función SDL_QuitSubSystem(Uint32 flags). Por ejemplo, si quisiéramos desactivar el subsistema de cdrom usaríamos la siguiente línea:

```
SDL_QuitSubSystem(SDL_INIT_CDROM);
```

Video

La primera tarea que tenemos que acometer antes de empezar a mostrar información gráfica en la pantalla es establecer el modo de video. SDL nos lo pone fácil con la función `SDL_SetVideoMode()`. La función tiene la siguiente forma:

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

Esta función devuelve un puntero del tipo SDL_Surface. En SDL la "surface" o superficie es el elemento básico que utilizamos para construir los gráficos de nuestro juego. Imagina una superficie como un rectángulo que contiene información gráfica, por ejemplo de nuestra nave. La única superficie que es visible, es decir, que vemos en la pantalla del ordenador, es ésta que nos devuelve la función `SDL_SetVideoMode()`.

Los parámetros `witdth` y `height` son la anchura y la altura de la superficie en píxeles. Es decir, la resolución de la pantalla. En teoría aquí podemos poner cualquier cosa, pero evidentemente el hardware ha de soportar dicha resolución. Si la resolución no es soportada, SDL intentará poner una resolución válida lo más cercana a la requerida. Un poco más adelante veremos una función que nos permitirá conocer cuales son los modos válidos. Con el parámetro `bpp` le indicamos a SDL cuántos bits por pixels queremos establecer, es decir, la profundidad de color. Valores típicos son 8, 16, 24 y 32. Hoy en día no tiene sentido trabajar con 8, ya que sólo nos proporciona 256 colores posibles, además, habría que trabajar con paletas en vez de con color real. En este libro no vamos a cubrir el trabajo con paletas, ya que es tedioso y dudo que realmente lo necesites alguna vez.

El último parámetro es `flags`, que es un campo de bits. Estos son los posibles valores y su significado.

`SDL_SWSURFACE`	Crea la superficie de vídeo en la memoria principal
`SDL_HWSURFACE`	Crea la superficie en la memoria de vídeo.
`SDL_ASYNCBLIT`	Modo de blit asíncrono. Mejora el rendimiento en máquinas con más de un procesador, unque puede disminuirlo en máquinas con una sólo procesador.
`SDL_ANYFORMAT`	Fuerza el uso de los bpp de la surface actual. Hay que usarlo cuando queramos crear la superficie en una ventana.
`SDL_HWPALETTE`	Da a SDL acceso exclusivo a la paleta de color.
`SDL_DOUBLEBUF`	Sólo válido con SDL_HWSURFACE. Permite el uso del doble buffer.
`SDL_FULLSCREEN`	Intenta poner el modo a pantalla completa.
`SDL_OPENGL`	Crea un contexto OpenGL en la superficie.
`SDL_OPENGLBLIT`	Igual que la anterior, pero permite que SDL haga el render 2D.
`SDL_RESIZABLE`	Crea un ventana que puede cambiar de tamaño.
`SDL_NOFRAME`	Crea una ventana pero sin borde.

He aquí un ejemplo práctico del uso de SDL_SetVideMode().

```
SDL_Surface *screen;

screen = SDL_SetVideoMode(640,480,24,SDL_SWSURFACE| SDL_DOUBLEBUF);
if ( screen == NULL ){
    fprintf(stderr, "No se puede establecer el modo \
            de video 640x480: %s\n", SDL_GetError());
 exit(1);
}
```

Le hemos pedido a SDL que establezca un modo de video con una resolución de 640x480 píxeles y 24 bits de color (true color). Además le solicitamos que use la memoria de vídeo y una estrategia de doble buffer para repintado.

Antes hemos dicho que la superficie que nos devuelve `SDL_SetVideoMode` es la única visible. Sin embargo, es seguro que necesitaremos crear y utilizar superficies intermedias para almacenar gráficos, construir escenas, etc... Estas superficies no pueden ser directamente mostradas por pantalla, pero sí copiadas en la superficie principal (visible). A este proceso lo llamamos *bit blitting* y veremos más abajo cómo se realiza esta operación. La manera de crear una nueva superficie es mediante la función `SDL_CreateRGBSurface` que crea una superficie vacía y lista para usar. El formato de esta función es

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

A simple vista parece algo complicada, así que vamos a detenernos un poco en ella. Como puedes observar, esta función nos devuelve un puntero a una superficie, al igual que hacía la función `SDL_SetVideoMode`. Los parámetros `width` y `height` tienen el mismo significado que en SDL_SetVideoMode, y `depth` es lo mismo que `bpp` (me pregunto por qué no han mantenido la misma nomenclatura). Los posibles valores para flags son:

`SDL_SWSURFACE`	Crea la superficie de vídeo en la memoria principal
`SDL_HWSURFACE`	Crea la superficie en la memoria de vídeo.
`SDL_SRCCOLORKEY`	Permite el uso de transparencias (color key).
`SDL_SRCALPHA`	Activa el alpha-blending.

Un poco más adelante veremos más claramente el significado de las dos última opciones. Nos queda `Rmask`, `Gmask`, `Bmask` y `Amask`. Estos parámetros son un poco más complejos de comprender sin entrar en mucho detalle en las estructuras de datos internas de SDL, en concreto en la estructura `SDL_PixelFormat`. Baste decir que la R, la G, la B y la A con la que empiezan cada uno de estos parámetros vienen de Red, Green, Blue y Alpha respectivamente. `Rmask`, por ejemplo, es la mascara de bits que representa al color rojo puro. Recuerda que un color cualquiera viene dado por la mezcla de los colores rojo, verde y azul, por lo tanto, `Rmask` serían los bits que es necesario activar para conseguir el color que tiene un 100% de rojo, un 0% de verde y un 0% de azul. Lo mismo es válido para el resto de parámetros. Veamos un ejemplo de uso de esta función.

```
rmask = 0xff000000;
gmask = 0x00ff0000;
bmask = 0x0000ff00;
amask = 0x000000ff;
surface = SDL_CreateRGBSurface(SDL_SWSURFACE, 640, 480, 24,rmask, gmask,
bmask, amask);
if(surface == NULL) {
    printf("Error al crear la superficie");
```

```
    exit(1);
}
```

Cargando y mostrando gráficos

Vale, todo esto está muy bien, pero ¿cuándo vamos a empezar a ver algo?

Está bien. Pasemos directamente a un ejemplo completo. La figura3.1 muestra el resultado de ejecutar el programa ejemplo3_1.

Figura3.1.

```
/*************************************************************
 Ejemplo3_1
 (C) 2003 by Alberto García Serrano
 Programación de videojuegos con SDL
 *************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <SDL.h>


int main(int argc, char *argv[]) {

 SDL Surface *image, *screen;
 SDL Rect dest;
 SDL Event event;
 int done = 0;

   atexit(SDL_Quit);

   // Iniciar SDL
   if (SDL Init(SDL INIT VIDEO) < 0) {
      printf("No se pudo iniciar SDL: %s\n",SDL GetError());
      exit(1);
   }

   // Activamos modo de video
   screen = SDL SetVideoMode(640,480,24,SDL HWSURFACE);
   if (screen == NULL) {
      printf("No se puede inicializar el modo gráfico: \n",SDL_GetError());
      exit(1);
   }


   // Cargamos gráfico
   image = SDL_LoadBMP("nave.bmp");
    if ( image == NULL ) {
        printf("No pude cargar gráfico: %s\n", SDL GetError());
        exit(1);
    }

   // Definimos donde dibujaremos el gráfico
   // y lo copiamos a la pantalla.
   dest.x = 100;
   dest.y = 100;
   dest.w = image->w;
   dest.h = image->h;
   SDL_BlitSurface(image, NULL, screen, &dest);

   // Mostramos la pantalla
   SDL Flip(screen);

   // liberar superficie
   SDL_FreeSurface(image);

   // Esperamos la pulsación de una tecla para salir
   while(done == 0) {
      while ( SDL_PollEvent(&event) ) {
         if ( event.type == SDL_KEYDOWN )
            done = 1;
      }
   }

   return 0;
}
```

Hay una gran cantidad de cosas en este código que aún no hemos visto, pero no querría seguir mostrándote una larga lista de funciones y ejemplos incompletos. Vamos a usar este programa para introducir nuevas capacidades de SDL.

La primera parte del programa debería ya resultarte familiar. En ella inicializamos SDL y también el modo gráfico. La primera función desconocida que encontramos es `SDL_LoadBMP()`. El formato de esta función es:

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

Su funcionamiento es realmente sencillo. Sólo tiene un parámetro, que es el nombre (y path) del un fichero gráfico en formato .bmp (bitmap). Si la función tiene éxito nos devuelve una superficie del tamaño del gráfico que acabamos de cargar, si no, devuelve NULL. En nuestro ejemplo, hemos declarado la variable `*image` de tipo `SDL_Surface` para que albergue la superficie cargada.

La siguiente función que nos llama la atención es:

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

Esta función copia zonas rectangulares de una superficie a otra. Vamos a usarla para copiar el gráfico que hemos cargado anteriormente en la superficie `image` en la superfivie visible (`screen`). Es decir, para mostrar el gráfico en la pantalla.

Si observamos los parámetros, hay dos de tipo `SDL_Surface` y otros dos del tipo `SDL_Rect`. El primer tipo ya lo conocemos, en concreto son los parámetros `src` y `dst`, que son la superficie fuente y la de destino respectivamente. El parámetro `srcrect` define la zona o porción rectangular de la superficie desde la que queremos copiar. `Dstrect` es la zona rectangular de la superficie de destino en la que vamos a copiar el gráfico. El tipo SDL_Rect tiene la siguiente forma:

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

Esta estructura define un rectángulo. Como seguramente adivinarás `x` e `y` son la posición de la coordenada superior izquierda del rectángulo mientras que `w` y `h` son la anchura y la altura del triangulo, respectivamente. Veamos en detalle como usamos esta estructura:

```
SDL_Rect dest;

dest.x = 100;
dest.y = 100;
dest.w = image->w;
dest.h = image->h;
SDL_BlitSurface(image, NULL, screen, &dest);
```

Hemos declarado la variable `dest` y a continuación hemos definido el área rectangular de destino al que vamos a copiar el gráfico. En nuestro ejemplo, vamos a copiar el gráfico en la posición (100,100) de la superficie de destino. Para conocer el tamaño de la superficie del gráfico que hemos cargado podemos acceder a los campos `w` y `h` de la estructura SDL_Surface de dicho gráfico.

Fíjate en el segundo parámetro, al que hemos dado el valor NULL. Con esto le indicamos a SDL que queremos copiar toda la superficie en la que se haya el gráfico.

Algún avispado lector se preguntará qué ocurre si los campos `w` y `h` de los parámetros `srcdest` y `dstrect` no coinciden. Realmente no ocurre nada, ya que los campos `w` y `h` de `dstrect` no se utilizan. Por último, decir que si la la copia tiene éxito la función devuelve el valor 0, y –1 en caso contrario.

Otra utilidad de la estructura `SDL_Rect` es la de dibujar rectángulos en la pantalla mediante la función `SDL_FillRect`. Ilustrémoslo con un ejemplo:

```
SDL_Rect dest;
dest.x = 0;
dest.y = 0;
dest.w = screen->w;
dest.h = screen->h;
SDL_FillRect(screen,&dest,SDL_MapRGB(screen->format,0,0,0));
```

Esta función dibuja un cuadrilátero en una superficie con el color indicado. En el ejemplo se dibuja un rectángulo de color negro del tamaño de la superficie (pantalla), o lo que es lo mismo, borra la pantalla. El formato de la función es:

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

El primer parámetro es la superficie de destino, el siguiente parámetro es de tipo `SDL_Rect` y define el rectángulo. Por último, hay que indicar el color de relleno. Un poco más adelante comprenderemos el uso de `SDL_MapRGB`.

Sigamos con la siguiente función:

```
int SDL_Flip(SDL_Surface *screen);
```

Si estamos usando doble buffer (como es el caso de nuestro ejemplo), esta función intercambia los buffers, es decir, vuelca el contenido del buffer secundario al principal. Si tiene éxito devuelve 0, si no, devuelve –1. Si no usáramos doble buffer o nuestro hardware no lo soportara, tendríamos que usar `SDL_UpdateRect(screen,0,0,0,0)` en lugar de `SDL_Flip()`. El formato de esta función es:

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

Su misión es asegurar que el área que especifiquemos (sus parámetros deben serte ya familiares) sea actualizada.

Como te veo con cara rara, quizás sea este el momento de explicar en qué consiste eso del doble buffer.

Imagina que tienes que volcar a la pantalla los gráficos de tu juego. Primero pondrías el fondo (calculando previamente que porción del fondo tienes que mostrar), mostrarías las naves enemigas (calculando su posición previamente) y después dibujarías al héroe de tu juego. Entre el primer y el último gráfico que dibujas en pantalla puede pasar un tiempo bastante grande (informáticamente hablando). Durante este tiempo, la imagen del monitor ha tenido tiempo de actualizarse varias veces (retrazo vertical). Desgraciadamente, cuando esto ocurre aparecen parpadeos y guiños (flicking) indeseados y muy molestos. Mediante la técnica del doble buffer, todo el pintado se hace en una pantalla virtual, y sólo cuando está todo pintado, volcamos su contenido a la pantalla real (flipping). Este proceso de volcado se hace además de forma sincronizada con el retrazo vertical del monitor para evitar los parpadeos.

Es importante (sobre todo en un lenguaje como C/C++) liberar los recursos que ya no vamos a necesitar. En nuestro ejemplo usamos la función `SDL_FreeSurface()` para liberar (borrar) la superficie que contiene la imagen cargada. Su formato es:

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

La misión de las siguientes líneas es esperar la pulsación de una tecla cualquiera antes de acabar. Veremos más en profundidad su significado cuando tratemos los eventos.

Efectos especiales: Transparencias y alpha-blending

El hecho de que seamos capaces de mostrar el grafico de un avión en la pantalla es un paso importante, pero seamos realistas: no es demasiado espectacular. De hecho, ese recuadro rojo alrededor del avión es bastante antiestético ¿no? Ciertamente sí. Alguien puede haber ya pensado en una solución (salomónica, me atrevería a apuntar), que es hacer que ese horrible recuadro rojo sea de color negro, al igual que el fondo. Realmente es una solución, si te gustan los juegos con fondos de pantalla algo monótonos. Como siempre SDL viene al rescate. Echemos un vistazo a la siguiente función:

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

Esta función nos permite designar un color en la superficie, que pasamos como parámetro, y que tendrá el comportamiento de un color transparente. Es decir, que no se pintará cuando hagamos el *blitting*. La función nos devuelve 0 si todo fue bién y –1 en otro caso. El parámetro `flag` sólo puede tomar tres valores posibles:

- 0 para desactivar una transparencia previamente activada.
- `SDL_SRCCOLORKEY` para indicar que el tercer parámetro de la función corresponde al color que queremos que sea transparente y
- `SDL_RLEACCEL` que permite usar codificación RLE en la superficie para acelerar el *blitting*.
-

Este último flag sólo puede usarse en combinación con `SDL_SRCCOLORKEY`, es decir, sólo se permiten tres valores posibles: 0, `SDL_SRCCOLORKEY`, `SDL_SRCCOLORKEY|SDL_RLEACCEL`.
El tercer parámetro es el color, que queremos que sea transparente. Hay que tener en cuenta que el color ha de estar en el mismo formato de píxel (píxel format) que la superficie. Lo habitual es utilizar la función `SDL_MapRGB` para conseguir el valor del color que buscamos. Veamos un ejemplo:

```
SDL_SetColorKey(image, SDL_SRCCOLORKEY|SDL_RLEACCEL, SDL_MapRGB(image-
>format,255,0,0));
```

Añade esta línea al ejemplo anterior después de la carga del gráfico. Si ejecutas ahora el programa veras el avión sin ese feo marco rojo alrededor.

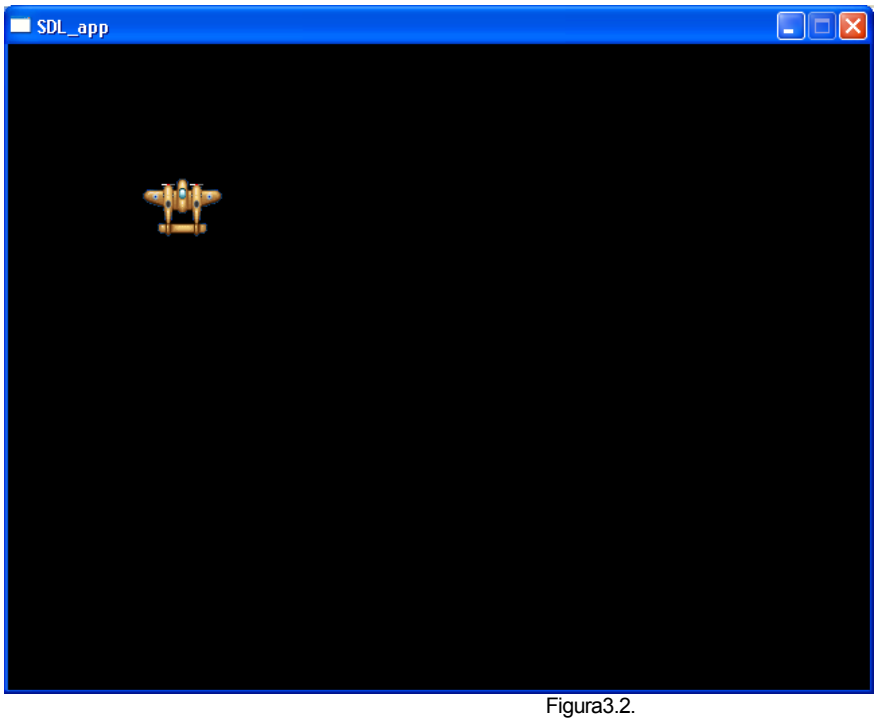


Figura3.2.

Lo que hemos hecho es, ni más ni menos, que decirle a SDL que el color rojo de la superficie que contiene el avión va a ser el color transparente. Como ves, esta función no tiene ningún secreto, aunque nos falta aclarar el funcionamiento de la función `SDL_MapRGB`.

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

Existe un campo en la estructura `SDL_Surface` (contiene la información de una superficie) llamado `format` que define el formato del píxel (píxel format) de la superficie. Dependiendo del formato que tenga el píxel (número de bits por pixels) la representación de un color concreto es diferente. Esta función nos devuelve el color solicitado en el formato de píxel que le pasamos como primer parámetro.
Observa que puedes obtener el formato de píxel de una superficie accediendo a su campo format (image->format). Los otros tres parámetros son las componentes de rojo, verde y azul (formato RGB) del color que queremos. En nuestro caso, el color rojo es (255,0,0).
Ahora que podemos usar colores transparentes, vamos a dar un paso más y vamos a utilizar una de las características más espectaculares de SDL: el alpha-blending o alpha para los amigos. Gracias al alpha-blending podemos hacer que una superficie sea translucida, es decir con cierto nivel de transparencia. Este nivel de transparencia tiene un rango de 0 a 255. Donde 0 es transparente y 255 es totalmente opaco. Como una imagen vale más que mil palabras, vamos a ver un nuevo ejemplo en la figura3.3.

```
/*****************************************
 Ejemplo3_2
 (C) 2003 by Alberto García Serrano
 Programación de videojuegos con SDL
 *****************************************/

#include <stdio.h>
#include <stdlib.h>
#include <SDL.h>


int main(int argc, char *argv[]) {

 SDL Surface *image, *screen;
 SDL Rect dest;
 SDL_Event event;
 int i, done = 0;

   atexit(SDL_Quit);

   // Iniciar SDL
   if (SDL_Init(SDL_INIT_VIDEO) < 0) {
      printf("No se pudo iniciar SDL: %s\n",SDL_GetError());
      exit(1);
   }

   // Activamos modo de video
   screen = SDL_SetVideoMode(640,480,24,SDL_HWSURFACE);
   if (screen == NULL) {
      printf("No se puede inicializar el modo gráfico: \n",SDL_GetError());
      exit(1);
   }


   // Cargamos gráfico
   image = SDL_LoadBMP("nave.bmp");
    if ( image == NULL ) {
        printf("No pude cargar gráfico: %s\n", SDL GetError());
        exit(1);
    }

   // Definimos color para la transparencia
   SDL SetColorKey(image,SDL SRCCOLORKEY|SDL RLEACCEL,SDL MapRGB(image->format,255,0,0));

   // Vamos a dibujar 100 graficos
   for (i=1 ; i<=100 ; i++) {
      // Ajustamos el canal alpha
      SDL_SetAlpha(image,SDL_SRCALPHA|SDL_RLEACCEL,rand() % 255);

      // Definimos donde dibujaremos el gráfico
      // y lo copiamos a la pantalla.
      dest.x = rand() % 640;
      dest.y = rand() % 480;
      dest.w = image->w;
      dest.h = image->h;
      SDL_BlitSurface(image, NULL, screen, &dest);
   }

   // Mostramos la pantalla
   SDL Flip(screen);

   // liberar superficie
    SDL_FreeSurface(image);

   // Esperamos la pulsación de una tecla para salir
   while(done == 0) {
      while ( SDL PollEvent(&event) ) {
         if ( event.type == SDL_KEYDOWN )
            done = 1;
      }
    }

   return 0;
}
```

El resultado de ejecutar el programa ejemplo3_2 es el siguiente:

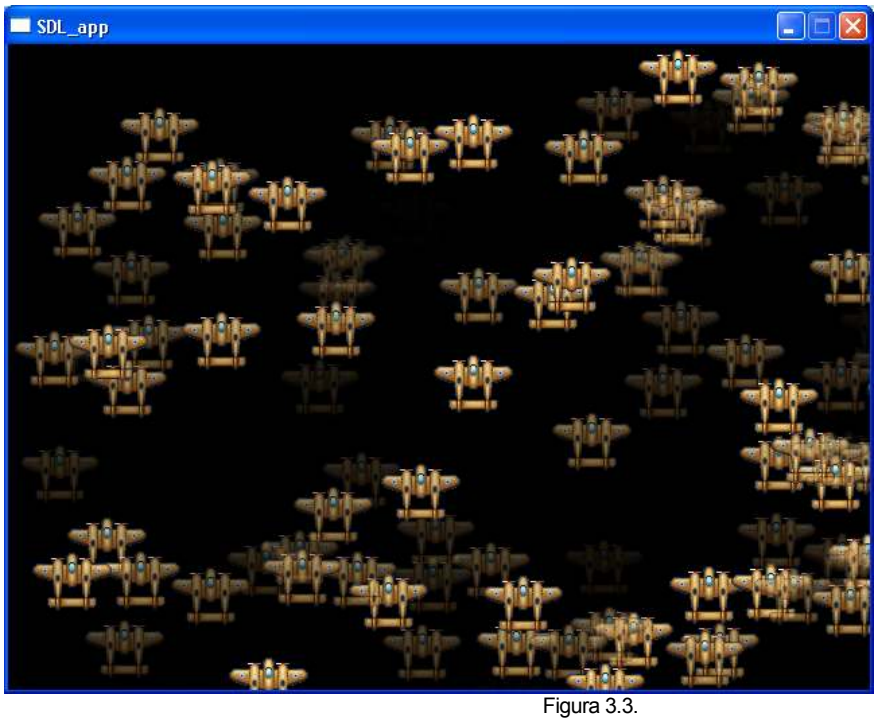


Figura 3.3.

La función encargada de realizar este trabajo tiene la siguiente forma:

```
int SDL_SetAlpha(SDL_Surface *surface, Uint32 flag, Uint8 alpha);
```

Como ves, es muy parecida a `SDL_SetColorKey`. En el caso del parámetro `flag`, los posibles valores son 0, para desactivar el alpha, `SDL_SRCALPHA` para indicar que el tercer parámetro de la función es el alpha que queremos aplicar a la superficie y `SDL_RLEACCEL` que se aplica junto a `SDL_SRCALPHA` para activar la aceleración RLE. El tercer parámetro es el nivel de alpha que queremos aplicar. Puede variar de 0 a 255. Esta función devuelve 0 si todo fue bien, y –1, en caso de error.

Otras funciones de interés

No quisiera terminar esta sección sobre el video en SDL sin presentarte algunas funciones más que sin duda antes o después te serán útiles.

Vamos a empezar por un par de funciones que nos van a permitir definir zonas de clipping o recorte. Veamos en que consiste el clipping con un ejemplo. Supongamos que vamos a hacer un simulador de vuelo. En la parte inferior de la pantalla estarán los mando del avión, en la parte superior habrá indicadores, y finalmente en la parte central de la pantalla está la zona donde se desarrolla la acción del juego. En cada ciclo de juego, hay que redibujar esta zona, pero no el resto de la pantalla (panel de control, indicadores, etc...). Con la función `SDL_SetClipRect` podemos decirle a SDL que cualquier operación gráfica que se realice quedará dentro de este rectángulo. Si el gráfico es mayor que este recuadro, se recorta. El formato de la función es:

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

El primer parámetro es la superficie sobre la que queremos efectuar el clipping, y el segundo ya debería serte familiar, es el rectángulo dentro del cual queremos permitir el volcado gráfico.

Añade las siguientes líneas al ejemplo anterior justo después de la inicialización del modo gráfico.

```
SDL_Rect clip_rect;

clip_rect.x=100;
clip_rect.y=100;
clip_rect.w=440;
clip_rect.h=280;
SDL_SetClipRect(screen ,&clip_rect);
```

Básicamente definimos un área rectangular de clipping que comienza en la posición (100,100) y tiene una anchura de 440 píxeles y una altura de 280 píxeles.

Figura3.4.

Para conocer qué área de clipping tiene una superficie podemos usar la siguiente función:

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

Como ves, el formato es idéntico al de `SDL_SetClipRect`, con la diferencia de que esta función rellenará la estructura `rect` con los datos del área de clipping.

La siguiente función en la que vamos a detenernos es la siguiente:

```
SDL_Surface *SDL_ConvertSurface(SDL_Surface *src, SDL_PixelFormat *fmt,
Uint32 flags);
```

Su misión es básicamente convertir una superficie al mismo formato de píxel que otra. Esto acelera las operaciones de blitting, ya que si dos superficies tienen diferente formato de píxel, SDL hace la conversión cada vez durante el blitting. El primer parámetro es la superficie que queremos convertir. El segundo, es el formato del píxel de la superficie a la que queremos convertir la primera. Como ya vimos antes, el formato del píxel de una superficie se encuentra en el campo `format` de la superficie (`superficie->format`). El tercer parámetro puede contener exactamente los mismo valores que la función `SDL_CreateRGBSurface` que vimos al principio de esta sección.

Por último (por ahora) quiero presentaros la siguiente función:

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

Su misión es devolvernos información sobre el modo de video actual. Si aún no se ha inicializado un modo de video, nos devuelve el "mejor" modo disponible (siempre según SDL). Esta función no tiene parámetros, y devuelve el resultado en una variable de tipo `SDL_VideoInfo`, que tiene la siguiente estructura:

```
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;
```

El significado de cada campo es el siguiente:

`Hw_available`	Indica si se pueden crear superficies en memoria de video.
`Wm_available`	Indica si hay un manejador de ventanas disponible.
`blit_hw`	Indica si el blitting hardware - hardware está acelerado.
`blit_hw_CC`	Indica si el blitting con transparencias hardware – hardware está acelerado.
`blit_hw_A`	Indica si el blitting con alpha hardware – hardware está acelerado.
`blit_sw`	Indica si el blitting software - hardware está acelerado.
`blit_sw_CC`	Indica si el blitting con transparencias software – hardware está acelerado.
`blit_sw_A`	Indica si el blitting con alpha software – hardware está acelerado.
`blit_fill`	Indica si el rellenado de color está acelerado.
`video_mem`	Cantidad de memoria total en Kilobites.
`Vfmt`	Puntero a la estructura SDL_PixelFormat que contiene el formato de píxel del sistema gráfico.

En el tintero nos dejamos muchas cosas sobre el video, pero ya tenemos las herramientas necesarias para comenzar a programar videojuegos. En la documentación de la librería SDL puedes encontrar información sobre muchos más tópicos que no hemos tratado aquí, como trabajo con paletas de color, acceso directo a las superficies, etc... Con los conocimientos que ahora tienes deberías poder seguir sin problema esta documentación.

Gestión de eventos

La gestión de eventos de SDL es realmente cómoda y fácil de usar. Seamos sinceros, si te las has visto con el sistema de eventos de Windows, SDL te va a parecer un juego de niños. El subsistema de eventos de SDL se inicializa junto al subsistema de video, por lo que no hay que inicializarlo expresamente. Todo esto está muy bien, pero ¿qué es un evento? Bien, en general, un evento es algo que sucede durante la ejecución del programa.

¿algo que sucede? ¿que respuesta es esa?
Vale, vale... espera que te explique. En SDL, un evento está asociado a una acción del usuario sobre la aplicación que se está ejecutando. Como sabes, para interactuar con un ordenador usamos periféricos de entrada, como un teclado, un ratón, etc...

Tipos de eventos

Teclado

El teclado es el principal dispositivo de entrada, aunque algunos se empeñen en que sea el ratón. Ahora, mientras tecleo estas líneas, lo estoy utilizando, y no puedo imaginar un dispositivo más cómodo para esta tarea.

Cada vez que pulses una tecla, SDL crea un evento que indica, no sólo que se pulsó una tecla, también cuál fue. Más concretamente SDL lanza un evento cada vez que se pulsa una tecla y también cuando de suelta. También nos informa sobre si hay alguna tecla especial pulsada como Ctrl, Alt o Shift.

Ratón

El otro gran dispositivo de entrada es el ratón, quizás más adecuado para ciertos tipos de juegos que el teclado. Hay dos tipos de eventos que devuelve el ratón, uno referente al movimiento y el otro referente al estado de los botones. En lo referente al movimiento, SDL nos informa sobre la posición del puntero, así como de la distancia que ha recorrido desde el último evento de ratón. En lo referente a los botones, el evento contiene información sobre si se ha pulsado o se ha soltado un botón del ratón, y por supuesto, cual de ellos fué.

Joystick

Sin duda, el joystick es el rey de los periféricos para jugar. Sin embargo en el PC no existe un estándar, y hay multitud de ellos. Debido a esto, SDL tiene un subsistema completo dedicado al joystick. Desgraciadamente, y quizás por no ser un periférico estándar, no está muy extendido. Los eventos que se generarán van a depender del joystick. En principio los principales son el movimiento del joystick y la pulsación de los botones. Algunos Joystick pueden generar otro tipo de eventos, pero vamos a centrarnos en estos dos, que son lo más comunes.

Otros eventos

También podemos conocer eventos relacionados con el sistema operativo, y que nos van a ser muy útiles. Sin duda el más importante es el que nos indica que se va a cerrar el juego (o que se va a cerrar la ventana del juego). Otros importantes son los referidos al cambio de tamaño de la ventana (si es que nuestro juego se está ejecutando en una ventana) o el que nos indica que debemos redibujarla (por cualquier motivo relacionado con el SO, como por ejemplo, que se superpuso una ventana de otra aplicación, etc...).

Lectura de eventos

Podemos interrogar al sistema por los eventos de tres formas posibles. Podemos esperar a que ocurra cierto evento para realizar una acción concreta, por ejemplo, un programa puede quedar a la espera de que pulsemos un botón en un cuadro de diálogo.

También podemos acceder al hardware directamente y leer el estado del dispositivo. Esto no te lo aconsejo a no ser que te guste hacer las cosas de la forma más difícil.

El tercer método es el llamado *polling*. Con este método los eventos se van almacenando, y pueden ser consultados en el momento que nos convenga. Esto cuadra muy bien con la forma en que funciona un juego. Acuérdate del *game loop*, en el que leíamos la entrada del jugador, la procesábamos, haciamos cálculos, movimientos o cualquier otra cosa y vuelta a empezar.

La función que nos permite leer el siguiente evento disponible es:

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

Esta función devuelve 1 si había un evento pendiente y 0 en caso contrario. Si hay algún evento disponible se almacena en el parámetro `event`. Como veremos a continuación, hay diferentes tipos de evento. Afortunadamente todos tiene un campo en común, que es el campo type (en realidad es una unión de C que contiene las diferentes estructuras para los diferentes eventos), por lo que podemos consultar el tipo de evento del que se trata. En el ejemplo3_1 y ejemplo3_2 puedes ver un ejemplo de cómo se utiliza esta función.

Eventos del teclado

Cada vez que se produce un evento de teclado (pulsar o soltar una tecla) este se almacena en la estructura `SDL_KeyboardEvent` que tiene la siguiente forma:

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

El campo `type` puede tomar los valores `SDL_KEYDOWN` o `SDL_KEYUP` para indicar si el evento corresponde a una pulsación o si por el contrario se soltó la tecla.

El campo `state` es redundante con el primero, ya que contiene exactamente la misma información, por lo que podemos ignorarlo tranquilamente. Para los más curiosos, los valores posibles de este campo son `SDL_PRESSED` o `SDL_RELEASED`.

El siguiente campo nos indica cual fue la tecla pulsada. Este campo es de tipo `SDL_keysym` que es una estructura que tiene el siguiente formato:

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

El primer campo es un código que genera el teclado (es decir, un código generado por el hardware). Puede cambiar de un sistema a otro, por lo que si quieres que tu juego pueda ejecutarse en otras plataformas será mejor que no lo uses. El campo `sym` es el más interesante de la estructura, ya que contiene un código identificativo de la tecla pulsada, pero al contrario que `scancode`, este código es generado por SDL y por lo tanto es igual en todas las plataformas . A continuación se muestra el listado de códigos `sym`.

SDLKey	ASCII
SDLK_BACKSPACE	'\b'
SDLK_TAB	'\t'
SDLK_CLEAR	
SDLK_RETURN	'\r'
SDLK_PAUSE	
SDLK_ESCAPE	'^['
SDLK_SPACE	' '
SDLK_EXCLAIM	'!'
SDLK_QUOTEDBL	'"'
SDLK_HASH	'#'
SDLK_DOLLAR	'$'
SDLK_AMPERSAND	'&'
SDLK_QUOTE	'''
SDLK_LEFTPAREN	'('
SDLK_RIGHTPAREN	')'
SDLK_ASTERISK	'*'
SDLK_PLUS	'+'
SDLK_COMMA	','
SDLK_MINUS	'-'
SDLK_PERIOD	'.'
SDLK_SLASH	'/'
SDLK_0	'0'
SDLK_1	'1'
SDLK_2	'2'
SDLK_3	'3'
SDLK_4	'4'
SDLK_5	'5'
SDLK_6	'6'
SDLK_7	'7'
SDLK_8	'8'
SDLK_9	'9'
SDLK_COLON	':'
SDLK_SEMICOLON	';'
SDLK_LESS	'<'
SDLK_EQUALS	'='
SDLK_GREATER	'>'
SDLK_QUESTION	'?'
SDLK_AT	'@'
SDLK_LEFTBRACKET	'['
SDLK_BACKSLASH	'\'
SDLK_RIGHTBRACKET	']'
SDLK_CARET	'^'
SDLK_UNDERSCORE	'_'
SDLK_BACKQUOTE	'`'
SDLK_a	'a'
SDLK_b	'b'
SDLK_c	'c'
SDLK_d	'd'
SDLK_e	'e'
SDLK_f	'f'

SDLKey	ASCII
SDLK_g	'g'
SDLK_h	'h'
SDLK_i	'i'
SDLK_j	'j'
SDLK_k	'k'
SDLK_l	'l'
SDLK_m	'm'
SDLK_n	'n'
SDLK_o	'o'
SDLK_p	'p'
SDLK_q	'q'
SDLK_r	'r'
SDLK_s	's'
SDLK_t	't'
SDLK_u	'u'
SDLK_v	'v'
SDLK_w	'w'
SDLK_x	'x'
SDLK_y	'y'
SDLK_z	'z'
SDLK_DELETE	'^?'
SDLK_KP0	
SDLK_KP1	
SDLK_KP2	
SDLK_KP3	
SDLK_KP4	
SDLK_KP5	
SDLK_KP6	
SDLK_KP7	
SDLK_KP8	
SDLK_KP9	
SDLK_KP_PERIOD	'.'
SDLK_KP_DIVIDE	'/'
SDLK_KP_MULTIPLY	'*'
SDLK_KP_MINUS	'-'
SDLK_KP_PLUS	'+'
SDLK_KP_ENTER	'\r'
SDLK_KP_EQUALS	'='
SDLK_UP	
SDLK_DOWN	
SDLK_RIGHT	
SDLK_LEFT	
SDLK_INSERT	
SDLK_HOME	
SDLK_END	
SDLK_PAGEUP	
SDLK_PAGEDOWN	
SDLK_F1	
SDLK_F2	
SDLK_F3	

SDLKey	ASCII
SDLK_F4	
SDLK_F5	
SDLK_F6	
SDLK_F7	
SDLK_F8	
SDLK_F9	
SDLK_F10	
SDLK_F11	
SDLK_F12	
SDLK_F13	
SDLK_F14	
SDLK_F15	
SDLK_NUMLOCK	
SDLK_CAPSLOCK	
SDLK_SCROLLOCK	
SDLK_RSHIFT	
SDLK_LSHIFT	
SDLK_RCTRL	
SDLK_LCTRL	
SDLK_RALT	
SDLK_LALT	
SDLK_RMETA	
SDLK_LMETA	
SDLK_LSUPER	
SDLK_RSUPER	
SDLK_MODE	
SDLK_HELP	
SDLK_PRINT	
SDLK_SYSREQ	
SDLK_BREAK	
SDLK_MENU	
SDLK_POWER	
SDLK_EURO	

El campo mod (de modificador) es un campo de bits que nos hablan del estado de ciertas teclas especiales como CTRL, ALT, SHIFT, etc... Su posibles valores son:

KMOD_NONE	Ningún modificador
KMOD_NUM	Numlock esta pulsado
KMOD_CAPS	Capslock está pulsado
KMOD_LCTRL	Control izquierdo está pulsado
KMOD_RCTRL	Control derecho está pulsado
KMOD_RSHIFT	Shift derecho está pulsado
KMOD_LSHIFT	Shift izquierdo está pulsado
KMOD_RALT	Alt derecho está pulsado
KMOD_LALT	Alt iquierdo está pulsado
KMOD_CTRL	Cualquier Control está pulsado
KMOD_SHIFT	Cualquier Shift está pulsado
KMOD_ALT	Cualquier Alt está pulsado

El último campo contiene el carácter unicode de la tecla pulsada. Por defecto este campo no se rellena, ya que la traducción conlleva cierta sobrecarga, por lo que asegúrate de activarlo sólo si lo necesitas. Para activar este campo usamos la siguiente función:

```
int SDL_EnableUNICODE(int enable);
```

Esta función devuelve el anterior estado antes del cambio. El parámetro `enable` pude tomar tres valores: 0 para desactivar la traducción unicode, 1 para activarla y –1 para dejar el estado como está. Esto nos sirve para conocer el estado mediante el valor de retorno de la función.

Si los 9 bits más altos del código están a 0, es un carácter ASCII. El siguiente código nos devuelve el carácter ASCII en la variable ch (en caso de que no sea un carácter unicode).

```
char ch;
if ( (keysym.unicode & 0xFF80) == 0 ) {
  ch = keysym.unicode & 0x7F;
}else {
  printf("Es un carácter unicode.\n");
}
```

Tanto en el ejemplo3_1, como en el ejemplo3_2 de la sección anterior sobre el video, puedes ver un ejemplo de lectura del teclado. Ahora deberías poder entender este código sin ningún problema.

Podemos, además de consultar el teclado mediante eventos, hacer una consulta directa al teclado para conocer su estado actual. Es como hacer una “fotografía” del estado del teclado en un momento dado. La siguiente función realiza esta tarea:

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

Esta función nos devuelve un puntero a un array con el estado de cada una de las teclas del teclado. En el parámetro `numkeys` se devuelve el tamaño de este array. Normalmente le pasaremos el valor NULL como parámetro. Para consultar este array utilizamos las mismas constantes de teclado que vimos antes. Si la tecla estaba pulsada

el valor almacenado en la posición del array correspondiente a la tecla es 1, si no estaba pulsada, será 0. Veámoslo con un ejemplo.

```
Uint8 *keys;
keys=SDL_GetKeyState(NULL);

if (keys[SDLK_UP] == 1) {arriba();}
if (keys[SDLK_DOWN] == 1) {abajo();}
if (keys[SDLK_LEFT] == 1) {izquierda();}
if (keys[SDLK_RIGHT] == 1) {derecha();}
if (keys[SDLK_LSHIFT] == 1) {disparo();}
```

Eventos de ratón

Tal y como comentábamos al inicio de esta sección, el ratón puede proporcionar dos tipos de eventos. El referente al movimiento y el referente a la pulsación de los botones.

Cuando sucede un evento de movimiento de ratón, el subsistema de eventos nos devuelve una estructura de tipo `SDL_MouseMotionEvent`.

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

El primer campo ya lo conocemos. Su único posible valor es `SDL_MOUSEMOTION`. El campo `state` devuelve el estado de los botones del ratón. Es una campo de bits que puede ser consultado cómodamente con la macro `SDL_BUTTON()`, pasándole como parámetro 1,2 o 3 para indicar botón izquierdo, central o derecho. `X` e `Y` son las coordenadas del ratón. `xrel` e `yrel` son la posición relativa respecto al último evento de ratón.

El otro tipo de evento relacionado con el ratón es el referente a la pulsación de los botones. Cuando se pulsa un botón del ratón se crea un evento del tipo `SDL_MouseButtonEvent`.

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

El campo `type` puede tomar los valores `SDL_MOUSEBUTTONDOWN` o `SDL_MOUSEBUTTONUP` para indicar si se pulsó o se soltó el botón. El campo `button` puede tomar los valores `SDL_BUTTON_LEFT`, `SDL_BUTTON_MIDDLE`, `SDL_BUTTON_RIGHT` para indicar que se pulsó (o se soltó) el botón izquierdo, central o el derecho. El campo `state` contiene la misma información que el campo `type`, por lo que podemos ignorarlo. Sus posibles valores son `SDL_PRESSED` o `SDL_RELEASED`. Por último, los campos `X` e `Y` contienen las coordenadas del ratón en el momento que se produjo el evento.

Eventos del joystick

En esta sección vamos a ver como el subsistema de eventos se comunica con el joystick. Cuando lleguemos a la sección del subsistema de gestión del joystick entraremos más en profundidad. Vamos a ver sólo dos de los eventos del joystick, el referente al movimiento del joystick y el referente a la pulsación de los botones. El primero de ellos genera un evento del tipo `SDL_JoyAxisEvent` que tiene la siguiente forma:

```
typedef struct{
  Uint8 type;
  Uint8 which;
  Uint8 axis;
  Sint16 value;
} SDL_JoyAxisEvent;
```

El campo `type` contine el valor `SDL_JOYAXISMOTION`. El segundo campo nos indica qué joystick produjo el evento, si es que tenemos más de uno conectado al ordenador. El campo `axis` nos dice qué eje se movió, y por último, `value` nos devuelve el valor de este movimiento dentro del rango que va de -32768 hasta 32767.

El otro tipo de evento relacionado con el joystick es `SDL_JoyButtonEvent`.

```
typedef struct{
  Uint8 type;
  Uint8 which;
  Uint8 button;
  Uint8 state;
} SDL_JoyButtonEvent;
```

El campo `type` puede tomar los valores `SDL_JOYBUTTONDOWN` o `SDL_JOYBUTTONUP`. Los dos campos siguientes son idénticos al del evento `SDL_JoyAxisEvent`. El último campo puede tomar los valores `SDL_PRESSED` o `SDL_RELEASED`.

Para poder utilizar la gestión de eventos del joystick hemos de activar esta opción antes con la siguiente función:

```
int SDL_JoystickEventState(int state);
```

Los posibles valores de `state` son `SDL_QUERY`, `SDL_ENABLE` o `SDL_IGNORE`. Con la primera consultamos el estado actual (activado o desactivado). Las otras dos opciones son para activar y desactivar la lectura del joystick mediante eventos. Si como parámetro le pasamos `SDL_QUERY` a la función, nos devolverá el estado actual, si no, nos devolverá el nuevo estado.

En la sección dedicada al joystick profundizaremos más.

Otros eventos

Uno de los eventos más importantes es `SDL_QuitEvent`. Este evento sucede cuando el sistema operativo quiere cerrar la aplicación (ya sea porque el usuario ha cerrado la ventana del juego o porque simplemente el SO lo decidió así por alguna oscura razón).

```
typedef struct{
  Uint8 type
} SDL_QuitEvent;
```

La estructura sólo tiene el campo `type`, que tomará el valor `SDL_QUIT`. Cuando se recibe este evento, hay que realizar las operaciones necesarias para cerrar la aplicación, como liberar memoria, guardar datos, etc...

El siguiente evento que vamos a ver es `SDL_ExposeEvent` ya que tiene exactamente la misma forma que `SDL_QuitEvent`.

```
typedef struct{
  Uint8 type
} SDL_ExposeEvent;
```

El campo `type` contendrá el valor `SDL_VIDEOEXPOSE`. Este evento se genera cuando el gestor de ventanas (Windows, KDE, etc...) ha realizado alguna modificación en las ventanas (ya sea por que el usuario ha movido alguna ventana o por que simplemente el gestor de ventanas a realizado un refresco) y nuestra aplicación necesita ser redibujada.

Relacionado también con el gestor de ventanas tenemos el evento `SDL_ResizeEvent`.

```
typedef struct{
  Uint8 type;
  int w, h;
} SDL_ResizeEvent;
```

Este evento se genera cuando la ventana cambia de tamaño (si es que el juego se está ejecutando en una ventana). Normalmente, si esto sucede tendremos que recalcular la posición de los elementos del juego. El campo `type` contiene el valor `SDL_VIDEORESIZE`. Los campos `w` y `h` son la nueva anchura y altura de la ventana.

Vamos a terminar con el evento `SDL_ActiveEvent`. En un entorno multiventana, el usuario puede pasar de una aplicación a otra. La forma de conocer si tenemos la atención del usuario es mediante este evento.

```
typedef struct{
  Uint8 type;
  Uint8 gain;
  Uint8 state;
} SDL_ActiveEvent;
```

El primer campo tendrá el valor `SDL_ACTIVEEVENT`. El campo `gain` valdrá 1 si la aplicación a retomado la atención o 0 si la perdió. Por último, pero no por ello, menos importante tenemos el campo `state`, que nos amplía más información sobre el campo `gain`. Sus posibles valores son:

`SDL_APPMOUSEFOCUS` si se ganó (o perdió, según el campo gain) el foco del ratón, es decir, si el ratón entró (o salió) del área de la ventana de nuestra aplicación.

`SDL_APPINPUTFOCUS` si se ganó (o perdió) el foco de entrada por teclado.

`SDL_APPACTIVE` si la aplicación fue maximizada (o minimizada).

Joystick

Ya hemos visto como se puede consultar el joystick mediante la consulta de eventos. En esta sección vamos a profundizar en el uso del joystick. Al igual que con el teclado, el joystick también puede ser consultado de forma directa. Además de conocer el estado del joystick, nos puede interesar conocer otras informaciones, como por ejemplo, cuantos joysticks hay conectados (si es que hay alguno), que características tiene, cuantos botones, etc...

Vamos a centrarnos exclusivamente en los ejes y en los botones del joystick. El concepto de eje es muy simple. Cuando movemos la palanca de mando hacia arriba o hacia abajo, estamos moviendo el joystick sobre el eje vertical. De la misma forma, al mover el mando de forma lateral lo movemos sobre el eje horizontal. Por lo tanto, un joystick clásico tiene dos ejes. Si en vez de un joystick clásico tenemos un paddle (recuerda el clásico juego de raquetas pong, con un mando giratorio que movía la raqueta), este sólo constará de un eje. Para poder utilizar el joystick desde SDL tenemos que inicializarlo con el flag SDL_INIT_JOYSTICK. Antes de entrar en las funciones que nos van a permitir consultar los estados del joystick, vamos a ver algunas dedicadas a mostrar información sobre él.

Recopilando información sobre el joystick

Lo primero que necesitamos saber es si hay algún joystick conectado al ordenador, y en su caso, cuántos hay. La siguiente función nos provee esta información:

```
int SDL_NumJoysticks(void);
```

El valor devuelto por esta función es el número de joystick conectados al ordenador. Otra información que puede sernos útil es la siguiente:

```
const char *SDL_JoystickName(int index);
```

Nos devuelve un puntero a una cadena que contiene el nombre del joystick o su driver. El parámetro `index` es el número de joystick que queremos consultar.

Una vez que conocemos el número de joysticks conectados al ordenador, es el momento de utilizarlos. Lo primero que hay que hacer es abrirlos.

```
SDL_Joystick *SDL_JoystickOpen(int index);
```

Como en la anterior función, `index` es el número del joystick que queremos abrir. La función devuelve un puntero del tipo `SDL_Joystick`. No vamos a entrar en detalle sobre esta estructura de datos. Simplemente saber que el tipo devuelto es el que usaremos en las siguientes funciones para referirnos al joystick que acabamos de abrir.

Como abras adivinado, SDL dispone de otra función para cerrar el joystick.

```
void SDL_JoystickClose(SDL_Joystick *joystick);
```

Como parámetro hay que pasarle el puntero que nos devolvió la función `SDL_JoystickOpen` al abrir el joystick. Es importante que cerremos el joystick antes de finalizar el programa (en realidad es importante que cerremos todo lo que inicialicemos en un programa).

Si tenemos más de un joystick conectados, nos puede resultar interesante saber cuáles están abiertos y cuáles no. La función `SDL_JoystickOpened` nos ofrece esta información:

```
int SDL_JoystickOpened(int index);
```

El parámetro `index` es el número de joystick que queremos consultar. Esta función devuelve 1 si el joystick consultado está abierto y 0 en caso contrario.

Nos queda ver un par de funciones vitales antes de entrar en harina.

```
int SDL_JoystickNumAxes(SDL_Joystick *joystick);
int SDL_JoystickNumButtons(SDL_Joystick *joystick);
```

Estas funciones nos devuelven el número de ejes y el número de botones respectivamente que tiene el joystick. Como parámetro le pasamos el joystick que queremos consultar (el valor devuelto por `SDL_JoystickOpen`).

Leyendo el joystick

Como comentamos al principio de la sección, nos vamos a centrar en la lectura de los ejes y de los botones. Si quieres profundizar, puedes remitirte a la documentación de SDL.

Antes de consultar el estado del joystick, hay que hacer una llamada a la función `SDL_JoystickUpdate`.

```
void SDL_JoystickUpdate(void);
```

Esta función, que no toma ni devuelve ningún parámetro, actualiza el estado de los joysticks abiertos. Antes de cada lectura de estado hay que hacer una llamada a esta función.

El estado de los ejes del joystick lo consultamos con la siguiente función:

```
Sint16 SDL_JoystickGetAxis(SDL_Joystick *joystick, int axis);
```

Como parámetros le indicamos el joystick y el eje que queremos consultar. Nos devolverá el valor del estado del eje. Este valor estará entre -32768 y 32768.
Para la consulta de los botones utilizaremos la función

```
Uint8 SDL_JoystickGetButton(SDL_Joystick *joystick, int button);
```

Esta función es similar a la anterior, sólo que en vez del eje a consultar, le pasamos el botón a consultar. El valor devuelto puede ser 1, el botón está pulsado y 0 en caso contrario.
El siguiente ejemplo (ejemplo3_3) muestra el uso del joystick.

```
/***************************************
 Ejemplo3_3
 (C) 2003 by Alberto Garcia Serrano
 Programación de videojuegos con SDL
 ***************************************/

#include <stdio.h>
#include <stdlib.h>
#include <SDL.h>


int main(int argc, char *argv[]) {

 SDL Surface *image, *screen;
 SDL Rect dest;
 SDL_Event event;
 Sint16 joyx,joyy;
 SDL_Joystick *joystick;
 int done = 0;
 int x,y,button;

   atexit(SDL_Quit);

   // Iniciar SDL
   if (SDL_Init(SDL_INIT_VIDEO|SDL_INIT_JOYSTICK) < 0) {
      printf("No se pudo iniciar SDL: %s\n",SDL GetError());
      exit(1);
   }

   // Activamos modo de video
   screen = SDL_SetVideoMode(640,480,24,SDL_HWSURFACE);
   if (screen == NULL) {
      printf("No se puede inicializar el modo gráfico: \n",SDL_GetError());
      exit(1);
   }


   // Activa el Joystick
   if (SDL_NumJoysticks() >= 1) {
      joystick = SDL_JoystickOpen(0);
      SDL_JoystickEventState(SDL_ENABLE);
   }


   // Cargamos gráfico
   image = SDL_LoadBMP("nave.bmp");
    if ( image == NULL ) {
        printf("No pude cargar gráfico: %s\n", SDL_GetError());
        exit(1);
    }

   // Definimos color para la transparencia
   SDL_SetColorKey(image,SDL_SRCCOLORKEY|SDL_RLEACCEL,SDL_MapRGB(image->format,255,0,0));

   x = y = 100;
   button = 0;

   while(done == 0) {

      // Borramos la pantalla
      dest.x=0;
      dest.y=0;
      dest.w=640;
      dest.h=480;
      SDL_FillRect(screen,&dest,SDL_MapRGB(screen->format,0,0,0));

      // Definimos donde dibujaremos el gráfico
      // y lo copiamos a la pantalla.
      dest.x = x;
      dest.y = y;
      dest.w = image->w;
      dest.h = image->h;
      SDL BlitSurface(image, NULL, screen, &dest);

      // Mostramos la pantalla
      SDL_Flip(screen);
```

```
      // lectura directa del joystick
      joyx = SDL_JoystickGetAxis(joystick, 0);
      joyy = SDL JoystickGetAxis(joystick, 1);

      if (joyy < -10) {y-=5;}
      if (joyy > 10) {y+=5;}
      if (joyx < -10) {x-=5;}
      if (joyx > 10) {x+=5;}

      // Lectura de eventos
      while ( SDL_PollEvent(&event) ) {

         // salir del programa si se pulsa el boton del joystick
         if ( event.type == SDL JOYBUTTONDOWN) {
            done = 1;
         }

         // salir del programa si se pulsa una tecla
         // o se cierra la ventana
         if ( event.type == SDL KEYDOWN || event.type == SDL QUIT)
            done = 1;
      }
    }

   // liberar superficie
    SDL FreeSurface(image);

   // cerramos el joystick
   if (SDL_JoystickOpened(0)) {
      SDL_JoystickClose(joystick);
   }

   return 0;
}
```

Este ejemplo muestra nuestro ya familiar avión, y además nos permite moverlo con el joystick por la pantalla. Para ilustrar las dos formas de acceder al joystick (mediante consulta de eventos o de forma directa), hacemos la consulta de los ejes (movimiento) mediante consulta directa y la lectura del botón del joystick la hacemos mediante la consulta de eventos. Para salir sólo hay que pulsar cualquier tecla o el botón del ratón.

Audio

Créeme, no querrás depender exclusivamente del subsistema de audio de SDL para el sonido de tu juego. En comparación con el resto de los subsistemas, el audio de SDL es bastante espartano (y no precisamente porque sea del sur del Peloponeso). Afortunadamente, hay una librería llamada *sdl_mixer* que nos va a hacer la vida más fácil. La vamos a utilizar en el juego que desarrollaremos en próximos capítulos, y a menos que alguien tenga una acentuada tendencia masoquista, la mayor parte de los juegos creados con SDL la usan. Vamos a pasar de puntillas sobre este subsistema, y en el próximo capitulo presentaremos *sdl_mixer* y sus virtudes.
Empecemos por la estructura de datos `SDL_AudioSpec`.

```
typedef struct{
  int freq;
  Uint16 format;
  Uint8 channels;
  Uint8 silence;
  Uint16 samples;
  Uint32 size;
  void (*callback)(void *userdata, Uint8 *stream, int len);
  void *userdata;
} SDL_AudioSpec;
```

Esta estructura especifica el formato de audio que vamos a utilizar. Veamos más detalladamente cada campo.

El primero es `freq`, que especifica la frecuencia (en Hertzios) de reproducción del sample. Valores habituales son 11025 (calidad telefónica), 22050 (calidad radiofónica) o 44100 (calidad CD).

El campo `format` especifica el formato del sample (bits y tipo). Los posibles valores son:

`AUDIO_U8`	Sample de 8 bits sin signo
`AUDIO_S8`	Sample de 8 bits con signo
`AUDIO_U16 o AUDIO_U16LSB`	Sample de 16 bits sin signo en formato little-endian.
`AUDIO_S16 o AUDIO_S16LSB`	Sample de 16 bits con signo en formato little-endian.
`AUDIO_U16MSB`	Sample de 16 bits sin signo en formato big-endian.
`AUDIO_S16MSB`	Sample de 16 bits con signo en formato big-endian.
`AUDIO_U16SYS`	AUDIO_U16LSB o AUDIO_U16MSB dependiendo del peso de tu sistema (big o little endían)
`AUDIO_S16SYS`	AUDIO_S16LSB o AUDIO_S16MSB dependiendo del peso de tu sistema (big o little endían)

El campo `channels` indica el número de canales de audio. Pueden ser 1 para mono y 2 para estereo.

El campo `silence`, es un valor calculado que representa el valor para silencio.

El siguiente campo es `samples`, y contiene el tamaño del buffer de audio en samples.

El campo `size`, es el tamaño del buffer, pero esta vez en bytes. Es un campo calculado, así que no debemos preocuparnos por el. El campo que sigue es un puntero a una función de retrollamada, que es la función llamada cuando es necesario reproducir el sonido. Efectivamente, tal y como estas pensando, somos nosotros los encargados de desarrollar esta función. Su cometido es rellenar el buffer contenido en el puntero `stream`, con una cantidad de bytes igual a `len`. En principio puede parecer que no es una tarea demasiado compleja ¿verdad? Esta bien, si quieres reproducir un solo sonido aislado es probable que no, pero el 99% de las veces necesitarás mezclar varios sonidos (por ejemplo, una música con una explosión, o simplemente un disparo y una explosión). Aunque SDL nos ofrece una forma de hacerlo, no es demasiado efectivo, de hecho, si quieres una calidad decente, tendrás que programar tu propia función de mezcla, y créeme, no querrás tener que hacerlo.

El campo `userdata` es el mismo que se le pasa a la función de retrollamada. Podemos pasar información a nuestra función en caso necesario.

Vayamos ahora con las funciones de manejo de audio.

```
int SDL_OpenAudio(SDL_AudioSpec *desired, SDL_AudioSpec *obtained);
```

Tal y como habrás adivinado, esta función abre el dispositivo de audio. Como parámetros tiene dos punteros de tipo `SDL_AudioSpec`. En el primero le solicitamos las especificaciones con las que queremos trabajar. Esta función intentará abrir el dispositivo de audio con las especificaciones que le pidamos, aunque esto no siempre será posible. En cualquier caso, `obtained` apuntará a las especificaciones finales que la función ha podido conseguirnos. Si todo fue bien, nos devolverá 0, en caso contrario, -1.

Como casi todo en SDL (y también en la vida cotidiana), todo lo que abrimos, hay que cerrarlo. La función encargada de esto es:

```
void SDL_CloseAudio(void);
```

Como no tiene ningún parámetro ni devuelve ningún valor, es una candidata perfecta para usarla con la función `atexit`.

Una vez abierto es dispositivo de audio, necesitamos una función para iniciar y para el audio. Aquí la tenemos:

```
void SDL_PauseAudio(int pause_on);
```

Cuando pasamos como parámetro el valor 0, activamos la reproducción de sonido. Si pasamos un 1, hacemos una pausa.

Por último veamos un par de funciones que nos permiten el trabajo con archivos en formato .wav:

```
SDL_AudioSpec *SDL_LoadWAV(const char *file, SDL_AudioSpec *spec, Uint8
**audio_buf, Uint32 *audio_len);

void SDL_FreeWAV(Uint8 *audio_buf);
```

La primera función carga un archivo en formato .wav en memoria. El primer parámetro es una cadena con el nombre del archivo. Si todo fue bien, nos devolverá un puntero a la estructura `SDL_AudioSpec` del fichero .wav. En `audio_buff` se encuentra la

información del sample, y en `audio_len` su tamaño. Como ves, todo listo para pasárselo a la función de retrollamada. La segunda función libera el espacio ocupado por el sample. El ejemplo3_4 nos aclara un poco la forma de utilizar el subsistema de audio. Lo que hace es reproducir un sonido aleatorio. Este código es muy sencillo y permite ver como utilizar la función de retrollamada. En el próximo capitulo aprenderemos como hacer sonar a SDL con la ayuda de SDL_mixer.

```
/*************************************
 Ejemplo3_4
 (C) 2003 by Alberto Garcia Serrano
 Programación de videojuegos con SDL
 *************************************/

#include <stdlib.h>
#include <sdl.h>

SDL_Surface* superficie;
SDL_Event event;
SDL_AudioSpec* deseado;
SDL_AudioSpec* obtenido;
int done;

// declaración de la función de retrollamada
void retrollamada(void* userdata,Uint8* buffer,int len);

int main(int argc, char* argv[]) {

   // Inicializamos SDL
   if (SDL_Init(SDL_INIT_VIDEO|SDL_INIT_AUDIO) < 0) {
      printf("No se pudo inicializar SDL.\n");
      exit(1);
   }

   // Inicializamos modo de video
   // (en Windows es necesario para que funcione el audio)
   if ( (superficie = SDL_SetVideoMode(640,480,0,SDL_ANYFORMAT)) == NULL) {
      printf("No se pudo inicializar la ventana.\n");
      exit(1);
   }


   // Alojamos espacio para almacenar las estructuras
   deseado=new SDL_AudioSpec;
   obtenido=new SDL AudioSpec;

   // especificaciones deseadas
   deseado->freq=11025;
   deseado->format=AUDIO_S16SYS;
   deseado->channels=1;
   deseado->samples=4096;
   deseado->callback=retrollamada;
   deseado->userdata=NULL;

   // abrimos el dispositivo de audio
   if(SDL_OpenAudio(deseado,obtenido)<0) {
      printf("No puedo abrir el dispositivo\n");
      delete deseado;
      delete obtenido;
      exit(1);
   }

   atexit (SDL CloseAudio);

   delete deseado;

   // Empieza a sonar...
   SDL PauseAudio(0);

   // esperamos a que cierren la aplicación
   done = 0;
   while (!done) {
      if(SDL_PollEvent(&event)==0)
         if(event.type==SDL QUIT) done=1;
   }

   delete deseado;

   return(0);
}


// Función de retrollamada
// Simplemente llena el buffer con información aleatoria.
void retrollamada(void* userdata,Uint8* buffer,int len) {
```

```
   int i;
   for(i=0 ; i<len ; i++)
      buffer[i]=rand()%256;
}
```

CD-ROM

¿Te imaginas poder reproducir la música de tu juego desde una pista de audio del mismo CD en el que está grabado? Esto es realmente sencillo de hacer, ya que SDL nos ofrece acceso al dispositivo CD-ROM de forma sencilla. Como el resto de subsistemas de SDL, hay que indicar que queremos utilizar el subsistema de CD-ROM al inicializar SDL. Hay dos estructuras de datos que nos van a proporcionar información sobre el CD.

```
typedef struct{
  int id;
  CDstatus status;
  int numtracks;
  int cur_track;
  int cur_frame;
  SDL_CDtrack track[SDL_MAX_TRACKS+1];
} SDL_CD;
```

El campo `id` es un identificador único que identifica a cada CD. El campo status es un tipo enumerado que contiene los posibles estado del CD.

`CD_TRAYEMPTY`	No hay ningún CD en el CD-ROM
`CD_STOPPED`	El CD está detenido
`CD_PLAYING`	El CD está reproduciendo
`CD_PAUSED`	El CD está en pausa
`CD_ERROR`	Hay un error

El siguiente campo `numtracks` contiene el número de pistas del CD. `cur_track` contiene la pista actual, y `cur_frame` es el frame actual dentro de la pista. Un frame es una medida interna del CD. Dado que el CD puede almacenar tanto audio como datos, no es posible usar como patrón de medida el tiempo o la capacidad. Un frame equivale aproximadamente a 2 Kilobytes. Por último tenemos un array con la información de cada pista. Una pista va descrita por la siguiente estructura:

```
typedef struct{
  Uint8 id;
  Uint8 type;
  Uint32 length;
  Uint32 offset;
} SDL_CDtrack;
```

El campo `id` es el número de la pista. El campo `type` puede tomar los valores `SDL_AUDIO_TRACK` o `SDL_DATA_TRACK`. El primero nos indica que es una pista de audio, y el segundo que es una pista de datos. Los otros dos campos nos indican el tamaño de la pista en frames, y la posición de inicio de la pista también en frames.
Antes de poder utilizar el CD, tenemos que tener alguna información al respecto, como el número de CDs o el nombre asociado al dispositivo. Los dos funciones siguientes nos posibilitan recopilar dicha información:

```
int SDL_CDNumDrives(void);

const char *SDL_CDName(int drive);
```

La primera función nos devuelve el número de CD-ROMs conectados al sistema. La segunda nos devuelve una cadena con el nombre del CD-ROM. Como parámetro recibe el número del CD-ROM que queremos consultar.
Como era de esperar, antes de poder utilizar el CD-ROM hay que abrirlo. La función encargada de esto es la siguiente:

```
SDL_CD *SDL_CDOpen(int drive);
```

Como parámetro le pasamos el CD-ROM que queremos abrir, y nos devolverá un puntero a una estructura `SDL_CD` con la información del CD. La función que cierra el CD-ROM es

```
void SDL_CDClose(SDL_CD *cdrom);
```

Como parámetro le pasamos el valor devuelto por `SDL_CDOpen` al abrir el CD.
La función que nos permite conocer el estado del CD es `SDL_CDStatus`.

```
CDstatus SDL_CDStatus(SDL_CD *cdrom);
```

Nos devolverá el estado actual del CD-ROM. Los posibles valores son los mismos que puede tomar el campo `status` de la estructura `SDL_CD`.
Las siguientes dos funciones que vamos a presentar nos permiten reproducir pistas del CD.

```
int SDL_CDPlay(SDL_CD *cdrom, int start, int length);

int SDL_CDPlayTracks(SDL_CD *cdrom, int start_track, int start_frame, int
ntracks, int nframes));
```

La primera función comienza la reproducción del CD en el frame indicado por `start` y durante los frames indicados por `length`. Si hay algún problema con la reproducción devuelve –1, si la reproducción tuvo éxito devolverá 0.
La función `SDL_CDPlayTracks` reproduce una o varias pistas. Con el parámetro `start_track` le indicamos la pista de inicio, y `numtracks` el número de pistas a reproducir. El parámetro `start_frame` es el frame, dentro de la pista, donde queremos iniciar la reproducción. Por último `nframes` es el número de frames de la última pista que queremos reproducir.
Seguramente querrás poder comenzar la reproducción en un determinado momento expresado en tiempo en vez de en frames. Puedes usar la siguiente fórmula que nos devuelve la duración en segundos.

Longitud en frames / CD_FPS

La constante CD_FPS contiene los frames por segundos del CD-ROM. Para finalizar, y antes de que veamos un ejemplo sobre el manejo del CD-ROM, vamos a ver cuatro funciones que requieren poca expliación:

```
int SDL_CDPause(SDL_CD *cdrom);
int SDL_CDResume(SDL_CD *cdrom);
int SDL_CDStop(SDL_CD *cdrom);
int SDL_CDEject(SDL_CD *cdrom);
```

La primera función pone en pausa la reproducción del CD, mientras que `SDL_CDResume` continúa con la reproducción. Si queremos parar la reproducción

usaremos la función `SDL_CDStop`, y con `SDL_CDEject` expulsaremos la bandeja del CD_ROM.

El ejemplo siguiente reproduce la primera pista del CD que se encuentre en la primera lectora del sistema.

```
/*****************************************
 Ejemplo3_5
 (C) 2003 by Alberto Garcia Serrano
 Programación de videojuegos con SDL
 *****************************************/

#include <stdlib.h>
#include <sdl.h>

SDL_Surface* superficie;
SDL_Event event;
SDL_CD *cdrom;
int pista = 0;
int done;

int main(int argc, char* argv[]) {

   // Inicializamos SDL
   if (SDL_Init(SDL_INIT_VIDEO|SDL_INIT_CDROM)<0) {
      printf("No se pudo inicializar SDL.\n");
      exit(1);
   }

   // Inicializamos modo de video
   if ( (superficie = SDL_SetVideoMode(640,480,0,SDL_ANYFORMAT)) == NULL) {
      printf("No se pudo inicializar la ventana.\n");
      exit(1);
   }

   // abrimos el primer CDROM del sistema
   if((cdrom = SDL_CDOpen(0))<0) {
      printf("No puedo abrir el dispositivo\n");
      exit(1);
   }

   // Status cd CD
   SDL CDStatus(cdrom);

   // iniciamos la reproducción de la primera pista del CD
   if ((SDL_CDPlay(cdrom, cdrom->track[pista].offset, cdrom->track[pista].length)) < 0) {
      printf("No puedo reproducir el CD\n");
      exit(1);
   }

   // esperamos a que cierren la aplicación
   done = 0;
   while (!done) {
      if(SDL_PollEvent(&event)==0)
         if(event.type == SDL QUIT) done=1;
   }

   // detenemos la reproducción y cerramos el CDROM
   SDL_CDStop(cdrom);
   SDL_CDClose(cdrom);
   return(0);
}
```

El Window Manager

Un *window manager* o gestor de ventanas es un programa que nos permite ejecutar aplicaciones dentro de ventanas, realizar acciones como cortar y pegar entre aplicaciones, mover y cambiar el tamaño de las ventanas, etc... En el caso de Windows, el gestor de ventanas forma parte del propio sistema operativo. En otros entornos, como Linux, podemos trabajar con el gestor de ventanas que más nos guste, como KDE o Gnome por nombrar los dos más conocidos. Habrás notado que todas las aplicaciones que hemos realizado hasta ahora, al ejecutarse mostraban el nombre SDL_app en la ventana. Este es el nombre por defecto. Podemos cambiar este nombre mediante la siguiente función:

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

Donde el primer parámetro es el título de la aplicación y el segundo el nombre del icóno. En Windows este segundo parámetro no se utiliza. Podemos conocer los valores actuales con la siguiente función.

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

Para cambiar el icono de la aplicación podemos utilizar la función

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

El primer parámetro es una surface que contiene el icóno. En Windows este icóno ha de tener una resolución de 32x32 píxeles. El segundo parámetro es un puntero a un array conteniendo la máscara de bits para el icóno (esto es usado para definir la forma del icóno y sus partes transparentes). Si como valor pasamos NULL, las transparencias estarán definias por el *color key* de la superficie. Veamos un ejemplo simple:

```
SDL_WM_SetIcon(SDL_LoadBMP("icon.bmp"), NULL);
```

Si queremos minimizar la aplicación mientras se ejecuta, usaremos la función

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

Si no se pudo minimizar la aplicación, esta función devolverá el valor 0.

Timming

Vamos a terminar esta introducción a SDL con el control de tiempo. Dejamos sin tratar todo lo referido a multitarea y programación de hilos. Hay que saber bién lo que se está haciendo y tener, al menos, unos sólidos conocimientos sobre multitarea y sistemas operativos para sacar provecho a esta capacidad de SDL. Afortunadamente, no nos va a ser necesaria para crear videojuegos.

Lo que si nos va a hacer falta es poder controlar el tiempo. Es una taréa esencial, ya que no todos los ordenadores funcionan a la misma velocidad. Si no controlaramos el tiempo, nuestro juego funcionaría bien en algunos ordenadores, pero en otros (los más rápidos) iría a una velocidad endiablada. Vamos con la primera función.

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

Esta función devuelve el número de milisegundos transcurridos desde que se inicializó SDL. Gracias a ella podemos controlar el tiempo transcurrido entre dos instantes dados dentro del programa, y por lo tan, controlar la velocidad del juego. Cuando desarrollemos el nuestro durante los próximos capítulos, podremos ver cómo controlar la velocidad de juego con esta función.

La función siguiente nos permite detener la ejecución del juego durante un tiempo determinado. Veamos que forma tiene.

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

Como parámetro le pasamos a la función el número de milisegundos que queremos esperar. Dependiendo del Sistema Operativo, la espera será más o menos exacta, pero por regla general el error puede ser de 10ms.

Capítulo 4

Librerías auxiliares para SDL

El uso de las facultades que me concedió la naturaleza es el único placer que no depende de la ayuda de la opinión ajena.
Ugo Foscolo.

Afortunadamente, además de las funciones que forman la librería SDL, podemos contar con una serie de librerías externas que nos van a ayudar bastante a la hora de programar nuestro juego. Las cuatro librerías auxiliares más habitules son SDL_image, SDL_mixer, SDL_ttf y SDL_net. En el juego de ejemplo de este libro utilizaremos SDL_mixer para la reproducción del sonido y la música, y SDL_ttf para el manejo de fuentes de letra. SDL_net es una librería que nos permite la conexión a redes TCP/IP, y es utilizada para crear juegos multijugador en red. SDL_image es una curiosa librería con una única función, que nos permite trabajar con múltiples formatos gráficos.

SDL_ttf

Es muy posible que hayas echado de menos alguna función en SDL para escribir texto en la pantalla gráfica. La librería SDL_ttf nos permite, básicamente, dibujar el texto que deseemos en una superficie utilizando la fuente de letra que queramos. La fuente de letra tiene que ser en formato ttf (true type font). Como ya viene siendo habitual, lo primero que tenemos que hacer es inicializar la librería.

```
int TTF_Init(void)
```

Esta función devolverá 0 si se realizó correctamente la inicialización y –1 en caso contrario. Como algún avispado lector ya habrá supuesto, la función inversa tiene la siguiente forma:

```
void TTF_Quit(void)
```

Al no recibir ni devolver esta función ningún valor, es una candidata perfecta para utilizarla junto con la función `atexit()`.

El siguiente paso es abrir la fuente que queremos utilizar, es decir, cargarla desde disco. Las dos siguientes funciones nos permiten abrir una fuente de letra.

```
TTF_Font * TTF_OpenFont(const char *file, int ptsize);
TTF_Font * TTF_OpenFontIndex(const char *file, int ptsize, long index)
```

Ambas funciones devuelven un puntero a un tipo `TTF_Font`. No es necesario que entremos en detalle sobre su estructura interna, ya que nos vamos a limitar a pasar este puntero a las funciones encargadas de realizar el pintado del texto. El primer parámetro es el fichero a cargar (una fuente ttf). El segundo es el tamaño de la fuente. En la segunda función, tenemos además un parámetro adicional que es la fuente que queremos usar en caso de que el archivo ttf contenga más de un tipo de letra. Cuando hayamos terminado de utilizar la fuente hemos de cerrarla con la siguiente función.

```
void TTF_Close(TTF_Font *font)
```

La función requiere poca explicación. Simplemente pasamos como parámetro la fuente que queremos cerrar.

El proceso de dibujar el texto en una superficie se llama *render*. Hay un total de 12 funciones dedicadas a este fin. Vamos a ver sólo 3 de ellas por ser las más habituales.

```
SDL_Surface * TTF_RenderText_Solid(TTF_Font *font, const char *text,
SDL_Color fg)

SDL_Surface * TTF_RenderText_Shaded(TTF_Font *font, const char *text,
SDL_Color fg, SDL_color bg)

SDL_Surface * TTF_RenderText_Blended(TTF_Font *font, const char *text,
SDL_Color fg)
```

Básicamente son iguales. Lo único que las diferencia es la calidad con la que es renderizado en texto. La primera función es la que ofrece menor calidad, y la última la que más. A mayor calidad, mayor tiempo necesita la función para realizar su cometido. Estas funciones devuelven un puntero a una superficie que contiene el texto dibujado. El parámetro `font` es un puntero a una fuente ya abierta con `TTF_OpenFont` o `TTF_OpenFontIndex`. El parámetro `text` es la cadena de texto que queremos imprimir en la pantalla. El parámertro `fg` es el color de la fuente y `bg` es el color de fondo. Obsérvese que los colores se pasan a la función en formato `SDL_Color`. Antes de ver un ejemplo completo del uso de SDL_ttf, vamos a ver un par de funciones que pueden sernos de utilidad.

```
int TTF_GetFontStyle(TTF_Font *font)

int TTF_SetFontStyle(TTF_Font *font, int style)
```

Estas funciones permiten conocer y seleccionar el estilo de texto que deseemos. Los posibles estilos son TTF_STYLE_BOLD para estilo **negrita**, TTF_STYLE_ITALIC, para estilo *itálica*, y TTF_STYLE_UNDERLINE para estilo subrayado. El estado normal es TTF_STYLE_NORMAL.

El siguiente programa es un ejemplo de uso de SDL_ttf.

```
/*****************************************************************************
 Ejemplo4_1
 (C) 2003 by Alberto Garcia Serrano
 Programación de videojuegos con SDL
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <SDL.h>
#include "SDL_ttf.h"


int main(int argc, char *argv[]) {

SDL Color bgcolor,fgcolor;
SDL_Rect rectangulo;
SDL_Surface *screen,*ttext;
TTF_Font *fuente;
const char texto[14]="Hola Mundo...";
char msg[14];
SDL Event event;
int done = 0;



   // Inicializamos SDL
   if (SDL Init(SDL INIT VIDEO) < 0) {
      printf("No se pudo iniciar SDL: %s\n",SDL_GetError());
      return 1;
   }

   // Inicializamos modo de video
   screen = SDL_SetVideoMode(640,480,24,SDL_HWSURFACE|SDL_DOUBLEBUF);
   if (screen == NULL) {
      printf("No se puede inicializar el modo gráfico: \n",SDL_GetError());
      return 1;
   }

   atexit(SDL_Quit);


   // Inicializamos SDL_ttf
   if (TTF Init() < 0) {
      printf("No se pudo iniciar SDL ttf: %s\n",SDL GetError());
      return 1;
   }

   atexit(TTF_Quit);

   // carga la funte de letra
   fuente = TTF_OpenFont("ariblk.ttf",20);

   // inicializa colores para el texto
   fgcolor.r=200;
   fgcolor.g=200;
   fgcolor.b=10;

   bgcolor.r=255;
   bgcolor.g=0;
   bgcolor.b=0;

   sprintf(msg,"%s",texto);
   ttext = TTF_RenderText_Shaded(fuente,msg,fgcolor,bgcolor);
   rectangulo.y=100;
   rectangulo.x=100;
   rectangulo.w=ttext->w;
   rectangulo.h=ttext->h;

   // Usamos color rojo para la transparencia del fondo
   SDL_SetColorKey(ttext,SDL_SRCCOLORKEY|SDL_RLEACCEL, SDL_MapRGB(ttext->format,255,0,0));

   // Volcamos la superficie a la pantalla
   SDL BlitSurface(ttext,NULL,screen,&rectangulo);

   // destruimos la fuente de letra
   TTF_CloseFont(fuente);
```

```
   // liberar superficie
    SDL_FreeSurface(ttext);

   // Esperamos la pulsación de una tecla para salir
   while(done == 0) {
      while ( SDL_PollEvent(&event) ) {
         if ( event.type == SDL_KEYDOWN )
            done = 1;
      }
    }

   return 0;
}
```

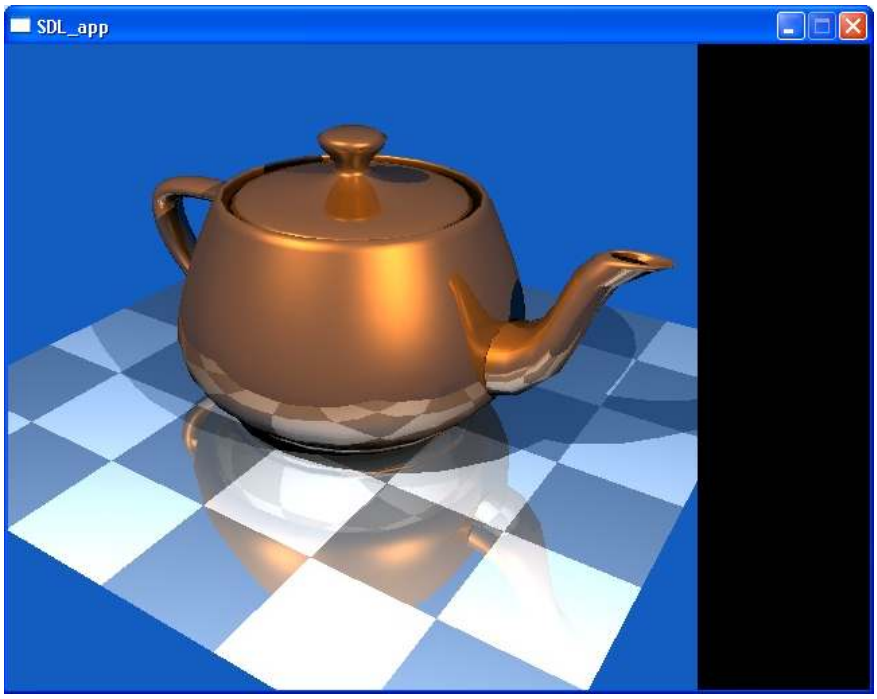
Esto es lo que deberíamos ver al ejecutar el programa de ejemplo. Lo primero que hacemos es cargar la fuente arial black. Usando la función `TTF_RenderText_Shade`, creamos una superficie con el color de fondo rojo. Antes de hacer el blitting a la pantalla, establecemos el color rojo como transparente.

SDL_image

La librería SDL_image es extremadamente simple, de hecho sólo tiene una función.

```
SDL_Surface * IMG_Load(const char *file)
```

Esta función tiene exactamente el mismo cometido que `SDL_LoadBMP`, es decir, cargar una imagen y almacenarla en una superficie. Recordemos, sin embargo, que `SDL_LoadBMP` sólo podía cargar archivos en formato BMP. Afortunadamente, la función IMG_Load puede manejar ficheros en formato BMP, PNM, XPM, LBM, PCX, GIF, JPG, PNG y TGA. Veamos a SDL_image en acción con un ejemplo.


SDL_app

```
/*****************************************************************************
 Ejemplo4_2
 (C) 2003 by Alberto Garcia Serrano
 Programación de videojuegos con SDL
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <SDL.h>
#include "SDL_image.h"


int main(int argc, char *argv[]) {

 SDL Surface *image, *screen;
 SDL_Rect dest;
 SDL_Event event;
 int done = 0;

   atexit(SDL Quit);

   // Iniciar SDL
   if (SDL_Init(SDL_INIT_VIDEO) < 0) {
      printf("No se pudo iniciar SDL: %s\n",SDL_GetError());
      exit(1);
   }

   // Activamos modo de video
   screen = SDL_SetVideoMode(640,480,24,SDL_HWSURFACE);
   if (screen == NULL) {
      printf("No se puede inicializar el modo gráfico: \n",SDL_GetError());
      exit(1);
   }


   // Cargamos gráfico
   image = IMG Load("teapot.jpg");
    if ( image == NULL ) {
        printf("No pude cargar gráfico: %s\n", SDL_GetError());
        exit(1);
    }

   // Definimos donde dibujaremos el gráfico
   // y lo copiamos a la pantalla.
   dest.x = 0;
   dest.y = 0;
   dest.w = image->w;
   dest.h = image->h;
   SDL BlitSurface(image, NULL, screen, &dest);

   // Mostramos la pantalla
   SDL_Flip(screen);

   // liberar superficie
    SDL FreeSurface(image);

   // Esperamos la pulsación de una tecla para salir
   while(done == 0) {
      while ( SDL_PollEvent(&event) ) {
         if ( event.type == SDL KEYDOWN )
            done = 1;
      }
    }

   return 0;
}
```

SDL_mixer

No voy a volver a recordarte lo tedioso que es trabajar con el subsistema de audio de SDL. SDL_mixer nos va a facilitar la tarea. Una de las mayores ventajas de SDL_mixer es que se encarga de realizar la mezcla de los canales de audio de forma automática. También puedes especificar tu propia función de mezcla, pero esto cae fuera del ámbito de este libro. SDL_mixer distingue entre la reproducción de sonidos y la reproducción de la música del juego (para la que reserva un canal exclusivo). Los formatos válidos para la música son WAV, VOC, MOD, S3M, IT, XM, Ogg Vorbis, MP3 y MIDI.

Para poder utilizar SDL_mixer, tendrás que inicializar el subsistema de audio de SDL, es decir, al inicializar SDL con `SDL_Init`, tendrás que incluir el flag `SDL_INIT_AUDIO`. Lo primero que hemos de hacer es inicializar la librería.

```
int Mix_OpenAudio(int frequency, Uint16 format, int channels, int chunksize)
```

Los párametros de esta función son similares a los de la estructura `SDL_AudioSpec`.

El primero es `freq`, que especifica la frecuencia (en Hertzios) de reproducción del sample. Valores habituales son 11025 (calidad telefónica), 22050 (calidad radiofónica) o 44100 (calidad CD).

El parámetro `format` especifica el formato del sample (bits y tipo). Los posibles valores son:

`AUDIO_U8`	Sample de 8 bits sin signo
`AUDIO_S8`	Sample de 8 bits con signo
`AUDIO_U16 o AUDIO_U16LSB`	Sample de 16 bits sin signo en formato little-endian.
`AUDIO_S16 o AUDIO_S16LSB`	Sample de 16 bits con signo en formato little-endian.
`AUDIO_U16MSB`	Sample de 16 bits sin signo en formato big-endian.
`AUDIO_S16MSB`	Sample de 16 bits con signo en formato big-endian.
`AUDIO_U16SYS`	AUDIO_U16LSB o AUDIO_U16MSB dependiendo del peso de tu sistema (big o little endían)
`AUDIO_S16SYS`	AUDIO_S16LSB o AUDIO_S16MSB dependiendo del peso de tu sistema (big o little endían)

El parámetro channels indica el número de canales de audio. Pueden ser 1 para mono y 2 para estereo. Para `chunksize`, el valor habitual es 4096 según la documentación oficial de SDL. En breve veremos lo que es un *chunk*.

Esta función devuelve –1 si hubo algún error, y 0 en caso de que todo vaya bien.
La función complementaria a `Mix_OpenAudio` es la siguiente:

```
void Mix_CloseAudio(void)
```

que recomiendo usar con `atexit()`. La misión de esta función, como bién debes saber, es finalizar la librería SDL_mixer.

Sonidos

Te estarás preguntando que es eso de un *chunk*. Básicamente un chunk es un sonido o efecto sonoro. SDL_mixer hace una abstracción y almacena cada efecto en un chunk. No vamos a entrar en detalles innecesarios, baste decir que la estructura de datos para almecenar estos chunks se llama `Mix_Chunk`. Una vez cargado un sonido en un chunk, la reproducción del sonido se llevará a cabo mediante un canal de audio, el cual, podremos especificar manualmente o dejar que SDL_mixer lo seleccione automáticamente. Cada canal puede reproducir simultáneamente un sólo sonido. Afortunadamente se soporta la reproducción de múltiples canales simultáneos, o lo que es lo mismo, de múltiples sonidos simultáneos.

Sin más preámbulos veamos esta bonita función:

```
Mix_Chunk *Mix_LoadWAV(char *file)
```

Sólo hay que indicarle a la función el archivo que queremos cargar. Si hubo algún error, la función devolverá NULL y en caso contrario tendremos un puntero a un tipo Mix_Chunk.

Cuando ya no necesitemos el sonido, es buena práctica liberar los recursos que utiliza. Lo hacemos con:

```
void Mix_FreeChunk(Mix_Chunk *chunk)
```

Basta con pasar como parámetro el chunk a liberar.
Respecto a los chunks, nos falta por ver una función.

```
int Mix_VolumeChunk(Mix_Chunk *chunk, int volume)
```

Efectivamente, esta función establece el volumen de reproducción del sonido. El volumen está dentro del rango de 0 a 128.
Ahora que tenemos el sonido cargado en un chunk, podemos reproducirlo mediante un canal de audio. La función encargada de tal menester es:

```
int Mix_PlayChannel(int channel, Mix_Chunk *chunk, int loops)
```

Esta función reproduce el sonido apuntado por el puntero chunk en el canal señalado por el parámetro `channel`. El parámetro `loops` indica cuántas veces ha de repetirse el sonido. Si sólo queremos reproducirlo una vez, pasamos 0 como valor. Con –1 se reproducirá indefinidamente. Si queremos que SDL_mixer selecciones el canal de forma automática (es lo mejor, a no ser que quieras utilizar grupos de canales) hay que pasar –1 como valor para el canal. Es conveniente informar a SDL_mixer de cuántos canales queremos utilizar. Lo hacemos con:

```
int Mix_AllocateChannels(int numchannels)
```

Lo normal es que le pasemos tantos canales como sonidos simultáneos queramos poder reproducir. Te aconsejo que no te quedes corto. Por otro lado, mientras más canales, más recursos requeríra el audio.

Además de `Mix_PlayChannel`, disponemos de tres funciones más para reproducir sonidos.

```
int Mix_PlayChannelTimed(int channel, Mix_Chunk *chunk, int loops, int
ticks)
```

```
int Mix_FadeInChannel(int channel, Mix_Chunk *chunk, int loops, int ms)

int Mix_FadeInChannelTimed(int channel, Mix_Chunk *chunk, int loops, int
ms, int ticks)
```

La primera función es idéntica a `Mix_PlayChannel`, con la diferencia de que el sonido se reproducirá durante los milisegundos indicados en el parámetro `ticks`.

La segunda función también realiza la reproducción del sonido, pero con un efecto de fade ascendente, es decir, el volumen del sonido irá aumentando desde 0 hasta el que corresponda al chunk de forma gradual. El tiempo que transcurrirá en ese aumento de volumen lo indicamos en con el parámetro `ms`. Este tiempo se expresa en milisegundos.

Por último, la tercera función no requiere demasiada explicación, ya que es la unión de las dos anteriores.

Mientras se reproduce un sonido, podemos pausarlo momentaneamente o pararlo. Las siguientes funciones nos permiten controlar la reproducción.

```
void Mix_Pause(int channel)

void Mix_Resume(int channel)

int Mix_HaltChannel(int channel)

int Mix_FadeOutChannel(int channel, int ms)
```

Los nombres de las funciones son bastante descriptivos. Todas aceptan como parámetro un canal. La primera función pone en estado de pausa el canal indicado, mientras que la segunda función reanuda su reproducción. Las dos últimas funciones paran la reproducción del canal, con la diferencia de que `Mix_HaltChannel` para la reproducción en seco y `Mix_FadeOutChannel` hace un efecto de fade contrario al de `Mix_FadeInChannel`. Esta última función, además del canal, acepta un segundo parámetro que es el tiempo en milisegundos que durará el efecto de fade.

A veces nos puede ser útil conocer el estado de un canal concreto. Las siguientes funciones nos ayudan en este cometido.

```
int Mix_Playing(int channel)

int Mix_Paused(int channel)
```

La primera función devolverá 1 si el canal que pasamos como parámetro se está reproduciendo actualmente y 0 en el caso de que esté en silencio. La segunda función devolverá 1 si el canal se está reproduciendo y 0 si está en pausa. El siguiente código de ejemplo reproduce un sonido y espera a que el usuario pulse una tecla para terminar.

```
/*****************************************************************************
 Ejemplo4_3
 (C) 2003 by Alberto Garcia Serrano
 Programación de videojuegos con SDL
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <SDL.h>
#include "SDL_mixer.h"


int main(int argc, char *argv[]) {

 SDL Surface *screen;
 SDL_Event event;
 Mix_Chunk *sonido;
 int done = 0;
 int canal;

   atexit(SDL Quit);

   // Iniciar SDL
   if (SDL_Init(SDL_INIT_VIDEO|SDL_INIT_AUDIO) < 0) {
      printf("No se pudo iniciar SDL: %s\n",SDL_GetError());
      exit(1);
   }

   // Activamos modo de video
   screen = SDL_SetVideoMode(640,480,24,SDL_HWSURFACE);
   if (screen == NULL) {
      printf("No se puede inicializar el modo gráfico: %s \n",SDL GetError());
      exit(1);
   }


   // Inicializamos SDL mixer
   if(Mix OpenAudio(22050, AUDIO S16, 2, 4096)) {
      printf("No se puede inicializar SDL_mixer %s\n",Mix_GetError());
      exit(1);
   }

   atexit(Mix CloseAudio);

   // Cargamos sonido
   sonido = Mix_LoadWAV("explosion.wav");
    if ( sonido == NULL ) {
        printf("No pude cargar sonido: %s\n", Mix_GetError());
        exit(1);
    }


   // Reproducción del sonido.
   // Esta función devuelve el canal por el que se reproduce el sonido
   canal = Mix PlayChannel(-1, sonido, 0);

   // Esperamos la pulsación de una tecla para salir
   while(done == 0) {
      while ( SDL_PollEvent(&event) ) {
         if ( event.type == SDL KEYDOWN )
            done = 1;
      }
    }

   // liberamos recursos
   Mix FreeChunk(sonido);

   return 0;
}
```

Música

Como dijimos anteriormente, SDL_mixer reserva un canal exclusivo para la música. Además, nos da soporte a múltiples formatos. Para cargar un archivo de música utilizamos la función `Mix_LoadMUS`.

```
Mix_Music *Mix_LoadMUS(const char *file)
```

Mediante el parámetro `file`, especificamos el archivo de música que queremos cargar. La función nos devuelve un puntero de tipo `Mix_Music`, que es parecido a `Mix_Chunk`, pero que nos permite almacenar música. Para liberar los recursos utilizamos la siguiente función.

```
void Mix_FreeMusic(Mix_Music)
```

La reproducción de la musica puede llevarse a cabo mediante alguna de las dos funciones siguientes.

```
int Mix_PlayMusic(Mix_Music *music, int loops)

int Mix_FadeInMusic(Mix_Music *music, int loops, int ms)
```

Estas dos funciones son muy similares a `Mix_PlayChannel` y `Mix_FadeInChannel`. De hecho, la única diferencia es que el primer parámetro es una música en lugar de un sonido. El resto de parámetros es exactamente igual.

El volumen de la música se puede establecer con la función `Mix_VolumeMusic`.

```
int Mix_VolumeMusic(int volume)
```

El rango válido para el volumen va de 0 a 128.

El control de la música puede llevarse a cabo con las funciones siguientes.

```
void Mix_PauseMusic()

void Mix_ResumeMusic()

int Mix_HaltMusic()

int Mix_FadeOutMusic(int ms)
```

Estas funciones han de serte ya familiares, ya que son similares a las utilizadas para controlar la reproducción de sonidos. Las dos primeras nos permiten pausar y reanudar la reproducción de la música. Observa que no tiene ningún parámetro ni devuelven ningún valor. Las dos útimas funciones paran la reproducción. Si hubo algún problema al intentar parar la reproducción devolverán –1, en caso contrario devolverán el valor 0. La última función además nos permite realizar la parada de la música de forma gradual (fadeout) teniendo que proveerle los milisegundos que deseamos que dure el fade.

La música, a diferencia de los sonidos simples, permiten mayor control. Además de reproducir, pausar y parar la música podemos situar la reproducción en el lugar que deseemos. Contamos con dos funciones que nos permiten realizar esta tarea.

```
void Mix_RewindMusic()

int Mix_SetMusicPosition(double position)
```

La primera función es muy sencilla. Su misisón es volver la reproducción de la música al principio.

La segunda función nos permite situar la posición de reproducción en el lugar deseado. La posición dependerá directamente del formato del archivo de música, pudiendo ser el tiempo en milisegundo para archivos de música digitalizada o la posición del patrón o el compás si es un archivo MOD o MIDI.

Vamos a terminar el capítulo de la música mostrando unas funciones que nos permiten conocer el estado del canal de música.

```
Int Mix_PlayingMusic()

Int Mix_PausedMusic()

Mix_MusicType Mix_GetMusicType(const Mix_Music *music)
```

Las dos primeras son similares a las utilizadas en la reproducción de sonidos vista en la sección anterior y no requieren mayor explicación. La tercera sí es nueva para nosotros. Nos permite conocer el formato del archivo musical que le pasamos como parámetro (o el que se está reproduciendo si `music` vale NULL). El valor devuelto es de tipo Mix_MusicType, y puede tomar los siguientes valores: MUS_WAV, MUS_MOD, MUS_MID, MUS_OGG, MUS_MP3. Si el valor devuelto es MUS_NONE, significa que no hay ninguna música en reproducción. Si el valor devuelto es MUS_CMD, significa que la reproducción no la está llevando a cabo SDL_mixer, sino un reproductor externo.

El siguiente código de ejemplo carga un archivo de música en formato MIDI y lo reproduce hasta que se pulse una tecla.

```
/*****************************************************************************
 Ejemplo4_4
 (C) 2003 by Alberto Garcia Serrano
 Programación de videojuegos con SDL
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <SDL.h>
#include "SDL_mixer.h"


int main(int argc, char *argv[]) {

 SDL Surface *screen;
 SDL_Event event;
 Mix_Music *musica;
 int done = 0;
 int canal;

   atexit(SDL Quit);

   // Iniciar SDL
   if (SDL_Init(SDL_INIT_VIDEO|SDL_INIT_AUDIO) < 0) {
      printf("No se pudo iniciar SDL: %s\n",SDL_GetError());
      exit(1);
   }

   // Activamos modo de video
   screen = SDL_SetVideoMode(640,480,24,SDL_HWSURFACE);
   if (screen == NULL) {
      printf("No se puede inicializar el modo gráfico: %s \n",SDL GetError());
      exit(1);
   }


   // Inicializamos SDL mixer
   if(Mix OpenAudio(22050, AUDIO S16, 2, 4096)) {
      printf("No se puede inicializar SDL_mixer %s\n",Mix_GetError());
      exit(1);
   }

   atexit(Mix CloseAudio);

   // Cargamos la música
   musica = Mix_LoadMUS("musica.mid");
    if ( musica == NULL ) {
        printf("No pude cargar musica: %s\n", Mix_GetError());
        exit(1);
    }


   // Reproducción la música.
   // Esta función devuelve el canal por el que se reproduce la música
   canal = Mix PlayMusic(musica, -1);

   // Esperamos la pulsación de una tecla para salir
   while(done == 0) {
      while ( SDL_PollEvent(&event) ) {
         if ( event.type == SDL KEYDOWN )
            done = 1;
      }
    }

   // paramos la música
   Mix HaltMusic();

   // liberamos recursos
   Mix_FreeMusic(musica);

   return 0;
}
```

Capítulo 5

Sprites: héroes y villanos

Enséñame un héroe y te escribiré una tragedia.
Francis Scott Fitzgerald.

Durante los capitulos siguientes vamos a profundizar en los diferentes aspectos concernientes a la programación de videojuegos. Ya dispones de las herramientas necesarias para emprender la aventura, así que siéntate cómodamente, flexiona tus dedos y prepárate para la diversión. Para ilustrar las técnicas que se describirán en los próximos capítulos, vamos a desarrollar un pequeño videojuego. Va a ser un juego sin grandes pretensiones, pero que nos va a ayudar a entender los diferentes aspectos que encierra este fascinante mundo. Nuestro juego va a consistir en lo que se ha dado en llamar *shooter* en el argot de los videojuegos. Quizás te resulte más familiar "matamarcianos". En este tipo de juegos manejamos una nave que tiene que ir destruyendo a todos los enemigos que se pongan en su camino. En nuestro caso, el juego va a estar ambientado en la segunda guerra mundial, y pilotaremos un avión que tendrá que destruir una orda de aviones enemigos. El juego es un homenaje al mítico 1942.

Figura 5.1. Juego 1942

Este capítulo lo vamos a dedicar a los sprites. Seguro que alguna vez has jugado a S*pace Invaders*. En este juego, una pequeña nave situada en la parte inferior de la pantalla dispara a una gran cantidad de naves enemigas que van bajando por la pantalla hacia el jugador. Pues bien, nuestra nave es un sprite, al igual que los enemigos, las balas y los escudos. Podemos decir que un sprite es un elemento gráfico determinado (una nave, un coche, etc...) que tiene entidad propia y sobre la que podemos definir y

modificar ciertos atributos, como la posición en la pantalla, si es o no visible, etc... Un sprite, pues, tiene capacidad de movimiento. Distinguimos dos tipos de movimiento en los sprites. El movimiento externo, es decir, el movimiento del sprite por la pantalla, y el movimiento interno o animación.

Para posicionar un sprite en la pantalla hay que especificar sus coordenadas. Es como el juego de los barquitos, en el que para identificar un cuadrante hay que indicar una letra para el eje horizontal (lo llamaremos eje Y) y un número para el eje horizontal (al que llamaremos eje X). En un ordenador, un punto en la pantalla se representa de forma parecida. Tenemos un eje horizontal (eje X) y otro vertical (eje Y). La esquina superior izquierda representa el centro de coordenadas. La figura siguiente muestra el eje de coordenadas en una pantalla con una resolución de 320 por 200 píxeles.

Figura 5.2. Ejes de coordenadas.

Un punto se identifica dando la distancia en el eje X al lateral izquierdo de la pantalla y la distancia en el eje Y a la parte superior de la pantalla. Las distancias se miden en píxeles. Si queremos indicar que un sprite está a 100 píxeles de distancia del eje vertical y 150 del eje horizontal, decimos que está en la coordenada (100,150).

Imagina ahora que jugamos a un videjuego en el que manejamos a un hombrecillo (piensa en juegos como *Pitfall* o *Renegade*). Al mover a nuestro hombrecillo, podemos observar cómo mueve las piernas y los brazos según avanza por la pantalla. Éste es el movimiento interno o animación. La siguiente figura muestra la animación del sprite de un gato.

Figura 5.3. Animación de un sprite.

Otra característica muy interesante de los sprites es que nos permiten detectar colisiones entre ellos. Esta capacidad es realmente interesante si queremos conocer cuando nuestro avión ha chocado con un enemigo o con uno de sus misiles.

Control de sprites

Vamos a realizar una pequeña librería (y cuando digo pequeña, quiero decir realmente pequeña) para el manejo de los sprites. Luego utilizaremos esta librería en nuestro juego. Por supuesto, también puedes utilizarla en tus propios juegos, así como ampliarla, ya que cubrirá sólo los aspectos básicos en lo referente a sprites.

Realizaremos la librería en C++ en lugar de C. Podríamos haberla creado completamente en C, pero considero conveniente introducir, aunque sea poco, algo de

programación orientada a objetos para que te vayas familiarizando. Observarás que es mucho más fácil reutilizar esta librería en tu propio código gracias a la POO. Consulta el apéndice B dedicado a C++ si no estás familiarizado con esto de los objetos y las clases. Prometo no utilizar más POO en el resto del libro (más de un lector lo lamentará, pero prefiero hacer el libro lo más accesible posible).

Dotaremos a nuestra librería con capacidad para movimiento de sprites, animación (un soporte básico) y detección de colisiones.

En el diagrama de clases de nuestra librería (Figura 5.4.) observamos que hay sólo dos clases. Si no sabes lo que es un diagrama de clases, te bastará saber que cada caja representa a una clase, y que la parte superior de la caja describe las variable miembro de la clase mientras que la parte inferior describe los métodos.

La clase `CFrame` será la encargada de contener un gráfico (superficie) independiente. Tiene un sólo miembro de clase, `img`, que será la encargada de almacenar la superficie, y dos métodos, `load` para cargar el gráfico y `unload` para liberar los recursos de la superficie. La razón por la que hemos utilizado dos clases es porque un mismo gráfico podría formar parte de dos sprites diferentes (imagina dos explosiones distintas que comparten los mismos fotogramas).

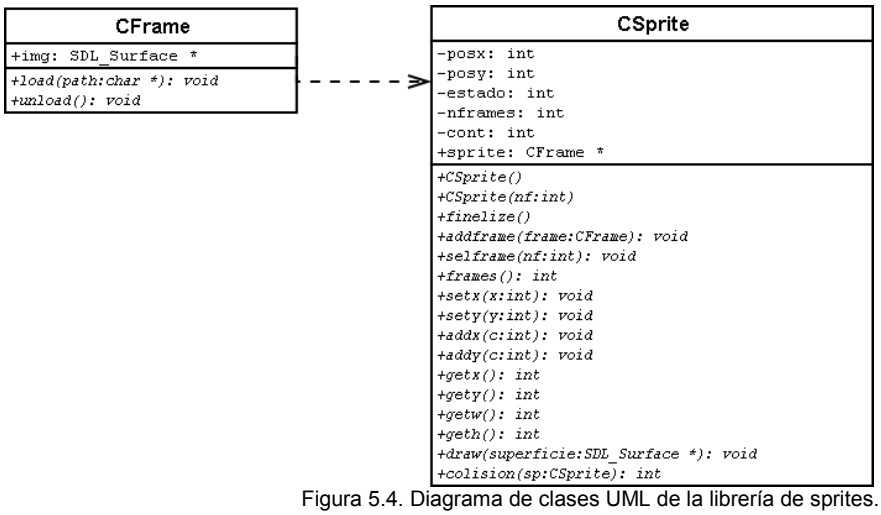


Figura 5.4. Diagrama de clases UML de la librería de sprites.

La clase `CSprite` representa a un sprite. Observa que tiene dos constructores (un ejemplo del polimorfismo en acción). Al crear un objeto de tipo `CSprite` podemos indicar el número de frames (fotogramas) que tendrá el sprite. Si creamos el objeto sin pasar ningún parámetro, el sprite tendrá un sólo frame. Añadiendo múltiples frames, podremos crear sprites animados. Una vez que ya no necesitamos nuestro sprite, podemos llamar al método `finalize()` que se encarga de llamar al método `unload()` de cada uno de los frames que forman parte del sprite, es decir, se encarga de liberar los recursos gráficos (superficies). La forma de añadir frames a nuestro sprite es mediante la función `addframe()`. Sólo tenemos que pasar como parametro el frame que queremos añadir a la animación del sprite. Con el método `selframe()` podemos seleccionar el frame actual, es decir, el que será dibujado. Para crear la sensación de movimiento interno, tendremos que ir dibujando nuestro sprite e ir cambiando el frame seleccionado cada vez. Para realizar la animación desde nuestro programa, hemos de conocer el número de frames que tiene el sprite. El método `frames()` nos informa sobre cuantos frames tiene el sprite actualmente.

Profundicemos en los métodos que nos permiten mover el sprite por la pantalla. Con los métodos `setx()` y `sety()` especificamos la posición del sprite en la pantalla. Con los métodos `addx()` y `addy()` podemos indicar un desplazamiento del sprite en pantalla tomando como punto de referencia la situación actual, es decir, `addx(5)` moverá el

sprite 5 píxeles en el eje a la derecha, mientras que `addx(-10)` lo moverá 10 píxeles a la izquierda.

Seguramente, necesitaremos en algún momento conocer información del sprite, como su posición o su tamaño. Los métodos `getx()` y `gety()` nos informarán sobre la posición en la que está el sprite. Los métodos `getw()` y `geth()` nos permiten conocer el tamaño del sprite en horizontal y en vertical respectivamente.

El método siguiente es `draw()`, y realizá la función más básica de un sprite, es decir, ¡dibujarlo! Este método simplemente dibuja el sprite en la posición y con el frame actual. Toda la información necesaria (posición, frame, etc...) está almacenada en miembros privados del objeto.

Por último, el método `colision()` nos permite comprobar si el sprite ha colisionado con otro. Sólo hemos de pasarle como parámetro el sprite con el que queremos comprobar la posible colisión.

Implementando los sprites

La clase `CFrame` es la encargada de cargar un gráfico que posteriormente será vinculado al sprite. Sus dos métodos son realmente sencillos.

```
// Método para cargar el frame
void CFrame::load(char *path) {
   img=SDL_LoadBMP(path);

   // Asignamos el color transparente al color rojo.
   SDL_SetColorKey(img,SDL_SRCCOLORKEY|SDL_RLEACCEL,
               SDL_MapRGB(img->format,255,0,0));
   img=SDL_DisplayFormat(img);

}

// Metodo para liberar el frame
void CFrame::unload(){
   SDL_FreeSurface(img);
}
```

El método `load()` carga un archivo en formato BMP mediante la función `SDL_LoadBMP`. Nuestros sprites van a tener como color transarente el rojo (255,0,0). Como es habitual, recurrimos a la función `SDL_SetColorKey` para tal fin. La función `SDL_DisplayFormat` se encarga de realizar una conversión del gráfico cargado al formato de la pantalla. Esto agiliza el proceso *blitting* al no tener que realizar conversiones cada vez que dibujamos el sprite.

El método `unload()` libera los recursos ocupados por el frame utilizando la función `SDL_FreeSurface`.

Fácil ¿no? Como puede observar, esta clase no tiene ningún misterio.

La clase `CSprite` es algo más grande, pero igual de sencilla. Cuando creamos un sprite, tenemos que indicar el número de frames que contendrá (o que podrá contener como máximo). Necesitamos hacer esto para reservar el espacio necesario en el array de frames. Vamos a colocar este código en el constructor de la clase.

```
CSprite::CSprite(int nf) {
   sprite=new CFrame[nf];
   nframes=nf;
   cont=0;
}
```

Además de reservar los elementos necesarios en el array de frames, inicializamos la variable `nframes` que contiene el número de frames máximo del sprite y la variable `cont`, que almacenará el número de frames que se van añadiendo.

Para liberar los recursos haremos uso del método `unload()` de la clase frame. Sólo hemos de llamar este método por cada frame del sprite.

```
CSprite::finalize() {
 int i;

   for (i=0 ; i<=nframes-1 ; i++)
      sprite[i].unload();
}
```

Ya estamos listos para ir añadiendo frames a nuestro sprite, el método `addframe()` toma como parámetro un frame y lo añade al array de frames. Antes de añadir el frame hemos de comprobar que no añadimos más frames que el máximo que permite el sprite (el valor que especificamos al crear el sprite). Por cada frame añadido incrementamos la variable `cont`.

```
void CSprite::addframe(CFrame frame) {
   if (cont<nframes) {
      sprite[cont]=frame;
      cont++;
   }
}
```

Los métodos que nos permiten establecer y consultar la posición del sprite son realmente simples. Las variables miembro de la clase que contienen este estado del sprite son `posx` y `posy`.

```
void setx(int x) {posx=x;}
void sety(int y) {posy=y;}
void addx(int c) {posx+=c;}
void addy(int c) {posy+=c;}
int getx() {return posx;}
int gety() {return posy;}
```

Para conocer el tamaño del sprite hemos de recurrir a consultar el tamaño del frame actual. Nótese que si un sprite está formado por frames de distintos tamaños (lo cual es posible, aunque no habitual), el tamaño devuelto será el del frame seleccionado.

```
int getw() {return sprite[estado].img->w;}
int geth() {return sprite[estado].img->h;}
```

Para seleccionar el frame actual, es decir, el que será dibujado en la próxima llamada al método `draw()`, utilizamos la variable miembro `estado`. Esta variable la utilizaremos para almacenar el frame actual o estado del sprite.

```
void CSprite::selframe(int nf) {
   if (nf<=nframes) {
      estado=nf;
   }
}
```

El método frames() simplemente consulta la variable miembro `cont`, que contiene el número de frames añadidos hasta el momento.

```
Int frames() {return cont;}
```

El método `draw()` utiliza la función `SDL_BlitSurface` para dibujar el sprite. Tenemos que proveer al método con la superficie sobre la que queremos realizar el blitting. De las variables miembro `posx` y `posy` obtenemos la posición en la que tenemos que dibujar el sprite, y de la variable `estado`, el frame que debemos dibujar.

```
void CSprite::draw(SDL_Surface *superficie) {
   SDL_Rect dest;

   dest.x=posx;
   dest.y=posy;
   SDL_BlitSurface(sprite[estado].img,NULL,superficie,&dest);
}
```

Nos resta dotar a nuestra librería con la capacidad de detectar colisiones entre sprites. La detección de colisiones entre sprites puede enfocarse desde varios puntos de vista. Imaginemos dos sprites, nuestro avión y un disparo enemigo. En cada vuelta del *game loop* tendremos que comprobar si el disparo ha colisionado con nuestro avión. Podríamos considerar que dos sprites colisionan cuando alguno de sus píxeles visibles (es decir, no transparentes) toca con un píxel cualquiera del otro sprite. Esto es cierto al 100%, sin embargo, la única forma de hacerlo es comprobando uno por uno los píxeles de ambos sprites. Evidentemente esto requiere un gran tiempo de computación, y es inviable en la práctica. En nuestra librería hemos asumido que la parte visible de nuestro sprite coincide más o menos con las dimensiones de la superficie que lo contiene. Si aceptamos esto, y teniendo en cuenta que una superficie tiene forma cuadrangular, la detección de una colisión entre dos sprites se simplifica bastante. Sólo hemos de detectar el caso en el que dos cuadrados se solapen.

Figura 5.5. Método simple de detección de colisiones

En la primera figura no existe colisión, ya que no se solapan las superficies (las superficies están representadas por el cuadrado que rodea al gráfico). La segunda figura muestra el principal problema de este método, ya que nuestra librería considerará que ha habido colisión cuando realmente no ha sido así. A pesar de este pequeño inconveniente, este método de detección de colisiones es el más rápido. Es importante que la superficie tenga el tamaño justo para albergar el gráfico. Este es el aspecto que tiene nuestro método de detección de colisiones.

```
int CSprite::colision(CSprite sp) {
int w1,h1,w2,h2,x1,y1,x2,y2;

   w1=getw();       // ancho del sprite1
   h1=geth();       // altura del sprite1
   w2=sp.getw();    // ancho del sprite2
   h2=sp.geth();    // altura del sprite2
```

```
  x1=getx();       // pos. X del sprite1
  y1=gety();       // pos. Y del sprite1
  x2=sp.getx();    // pos. X del sprite2
  y2=sp.gety();    // pos. Y del sprite2

  if (((x1+w1)>x2) && ((y1+h1)>y2) && ((x2+w2)>x1) && ((y2+h2)>y1)) {
    return TRUE;
  } else {
    return FALSE;
  }
}
```

Se trata de comprobar si el cuadrado (superficie) que contiene el primer sprite, se solapa con el cuadrado que contiene al segundo.

Hay otros métodos más precisos que nos permiten detectar colisiones. Voy a presentar un método algo más elaborado. Consiste en dividir el esprite en pequeñas superficies rectangulares tal y como muestra la próxima figura.

Figura 5.6. Un método más elaborado de detección de colisiones

Se puede observar la mayor precisión de este método. El proceso de detección consiste en comprobar si hay colisión de alguno de los cuadros del primer sprite con alguno de los cuadrados del segundo utilizando la misma comprobación que hemos utilizado en el primer método para detectar si se solapan dos rectangulos. Se deja como ejercicio al lector la implementación de este método de detección de colisiones (pista: tendrás que añadir a la clase sprite un array de elementos del tipo `SDL_Rect` que contenga cada uno de los cuadrados). A continuación se muestra el listado completo de nuestra librería. Está compuesta por dos archivos: el archivo de cabeceras *csprite.h* y el de implementación *csprite.cpp*.

csprite.h

```
#ifndef CSPRITE_H_
#define CSPRITE_H_

#define TRUE 1
#define FALSE 0

// CFrame representa un frame independiente de un sprite.
class CFrame {
public:
   SDL Surface *img;
   void load(char *path);
   void unload();
};


// La clase CSprite está formada por un array de frames;
class CSprite {
private:
   int posx,posy;
   int estado;
   int nframes;
   int cont;

public:
   CFrame *sprite;
   CSprite(int nf);
   CSprite();
   void finalize();
   void addframe(CFrame frame);
   void selframe(int nf);
   int frames() {return cont;}
   void setx(int x) {posx=x;}
   void sety(int y) {posy=y;}
   void addx(int c) {posx+=c;}
   void addy(int c) {posy+=c;}
   int getx() {return posx;}
   int gety() {return posy;}
   int getw() {return sprite[estado].img->w;}
   int geth() {return sprite[estado].img->h;}
   void draw(SDL_Surface *superficie);
   int colision(CSprite sp);
};

#endif /* CSPRITE H  */
```

csprite.cpp

```
#include <SDL.h>
#include "csprite.h"

// Sprite Class implementation

void CFrame::load(char *path) {
   img=SDL LoadBMP(path);

   // Asignamos el color transparente al color rojo.
   SDL_SetColorKey(img,SDL_SRCCOLORKEY|SDL_RLEACCEL, SDL_MapRGB(img->format,255,0,0));
   img=SDL DisplayFormat(img);

}

void CFrame::unload(){
   SDL_FreeSurface(img);
}

CSprite::CSprite(int nf) {
   sprite=new CFrame[nf];
   nframes=nf;
   cont=0;
}

CSprite::CSprite() {
int   nf=1;
   sprite=new CFrame[nf];
   nframes=nf;
   cont=0;
}

void CSprite::finalize() {
 int i;

   for (i=0 ; i<=nframes-1 ; i++)
      sprite[i].unload();
}

void CSprite::addframe(CFrame frame) {
   if (cont<nframes) {
      sprite[cont]=frame;
      cont++;
   }
}

void CSprite::selframe(int nf) {
   if (nf<=nframes) {
      estado=nf;
   }
}


void CSprite::draw(SDL Surface *superficie) {
   SDL Rect dest;

   dest.x=posx;
   dest.y=posy;
   SDL BlitSurface(sprite[estado].img,NULL,superficie,&dest);
}

int CSprite::colision(CSprite sp) {
int w1,h1,w2,h2,x1,y1,x2,y2;

   w1=getw();        // ancho del sprite1
   h1=geth();        // altura del sprite1
   w2=sp.getw();     // ancho del sprite2
   h2=sp.geth();     // alto del sprite2
   x1=getx();        // pos. X del sprite1
   y1=gety();        // pos. Y del sprite1
   x2=sp.getx();     // pos. X del sprite2
   y2=sp.gety();     // pos. Y del sprite2

   if (((x1+w1)>x2)&&((y1+h1)>y2)&&((x2+w2)>x1)&&((y2+h2)>y1)) {
      return TRUE;
```

```
   } else {
      return FALSE;
   }
}
```

Utilizando nuestra librería

En el ejemplo5_1 hay un listado completo de un programa que utiliza la librería que acabamos de desarrollar. Al ejecutarlo verá la siguiente ventana.

Podemos manejar el avión con las teclas del cursor o con el joystick. En la parte superior hay un avión enemigo parado. Si colisionamos con él acaba el programa.

```
/*****************************************************************************
 Ejemplo5_1
 (C) 2003 by Alberto Garcia Serrano
 Programación de videojuegos con SDL
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <SDL.h>
#include "csprite.h"


SDL_Surface *screen;
CFrame fnave;
CFrame fmalo;
CSprite nave(1);
CSprite malo(1);
SDL_Rect rectangulo;
SDL_Joystick *joystick;
int joyx, joyy;
int done=0;


// estructura que contiene la información
// de nuestro avión
struct minave {
   int x,y;
} jugador;


// Estructura que contiene información
// del avión enemigo
struct naveenemiga {
   int x,y;
} enemigo;

// Dibuja los esprites en la pantalla
void DrawScene(SDL Surface *screen) {

   // borramos el avión dibujado
   // en el frame anterior
   rectangulo.x=nave.getx();
   rectangulo.y=nave.gety();
   rectangulo.w=nave.getw();
   rectangulo.h=nave.geth();
   SDL_FillRect(screen,&rectangulo,SDL_MapRGB(screen->format,0,0,0));

   // dibuja avión
   nave.setx(jugador.x);
   nave.sety(jugador.y);
   nave.draw(screen);


   // Dibuja enemigo
   malo.setx(enemigo.x);
   malo.sety(enemigo.y);
   malo.draw(screen);

   // ¿ha colisionado con la nave?
   if (malo.colision(nave) == TRUE) {
      done=1;
   }

   // Mostramos todo el frame
   SDL_Flip(screen);
}


// Inicializamos estados
void inicializa() {
   jugador.x=300;
   jugador.y=300;
   enemigo.x=100;
   enemigo.y=100;
}

void finaliza() {
```

```
   // finalizamos los sprites
   nave.finalize();
   malo.finalize();

   // cerramos el joystick
   if (SDL_JoystickOpened(0)) {
      SDL_JoystickClose(joystick);
   }
}

// Preparamos los esprites
int InitSprites() {

   fnave.load("minave.bmp");
   nave.addframe(fnave);

   fmalo.load("nave.bmp");
   malo.addframe(fmalo);

   return 0;
}


int main(int argc, char *argv[]) {

SDL Event event;
Uint8 *keys;


   if (SDL_Init(SDL_INIT_VIDEO|SDL_INIT_JOYSTICK) < 0) {
      printf("No se pudo iniciar SDL: %s\n",SDL_GetError());
      return 1;
   }

   screen = SDL_SetVideoMode(640,480,24,SDL_HWSURFACE);
   if (screen == NULL) {
      printf("No se puede inicializar el modo gráfico: \n",SDL GetError());
      return 1;
   }

   atexit(SDL_Quit);

   inicializa();

   InitSprites();

   while (done == 0) {

      // dibujamos el frame
      DrawScene(screen);

      // consultamos el estado del teclado
      keys=SDL_GetKeyState(NULL);

      // consultamos el estado del joystick
      SDL_JoystickUpdate();

      joyx = SDL_JoystickGetAxis(joystick, 0);
      joyy = SDL_JoystickGetAxis(joystick, 1);

      if ((keys[SDLK UP] || joyy < -10) && (jugador.y > 0)) {jugador.y=jugador.y-(5);}
      if ((keys[SDLK_DOWN] || joyy > 10) && (jugador.y < 460)) {jugador.y=jugador.y+(5);}
      if ((keys[SDLK_LEFT] || joyx < -10) && (jugador.x > 0)) {jugador.x=jugador.x-(5);}
      if ((keys[SDLK_RIGHT] || joyx > 10) && (jugador.x < 620)) {jugador.x=jugador.x+(5);}


      while (SDL PollEvent(&event)) {

         if (event.type == SDL_QUIT) {done=1;}

         if (event.type == SDL_KEYDOWN || event.type == SDL_JOYBUTTONDOWN) {

            if (event.key.keysym.sym == SDLK ESCAPE) {
               done=1;
            }

         }
```

```
        }
    }


    finaliza();

    return 0;
}
```

En este ejemplo hay algunas diferencia con respecto a los anteriores. En la función `main()` hay un bucle while que se parece bastante a un *game loop*. Después de inicializar la posición de los aviones mediante la función `inicializa()` y cargar e inicializar los sprites en la función `InitSprites()`, entramos en un bucle en el que lo primero que hacemos es llamar a la función `DrawScene()`, que se encarga de dibujar los dos sprites en la pantalla. Acto seguido, consultamos el teclado y el joystick para comprobar si hay que realizar algún movimiento en nuestro avión, y en caso afirmativo, aplicamos dicho movimiento. La función `DrawScene()` es realmente la única que se encarga de dibujar los sprites. Esta función es llamada en cada iteración del *game loop*. Cada vez que se pintan los sprites, decimos que se ha dibujado un *frame*. Lo primero que hace es borrar el cuadrado que acupaba el avión en el frame anterior (realmente pintar un cuadro de color negro). Si no hiciéramos esto, el avión dejaría un feo rastro al moverse (puedes probar a quitar estas líneas y ver el efecto). Lo siguiente que hacemos es actualizar las coordenadas del sprite y dibujarlo con el método `draw()`. Hemos almacenado esta información en dos estructuras que contienen las coordenadas tanto de nuestro avión como del avión enemigo.

Despues del pintado, comprobamos la colisión de la nave enemiga con la nuestra. Si se produce la colisión, salimos del *game loop* poniendo la variable `done` a 1.

La clase `InitSprites()` también es interesante. En ella cargamos los frames que van a formar parte de los sprites (1 sólo frame en éste caso), y acto seguido lo añadimos al sprite.

Capítulo

6

Un Universo dentro de tu ordenador

Cuando quieres realmente una cosa, todo el Universo conspira para ayudarte a conseguirla.
Paulo Coelho.

En el segundo capítulo, ya desarrollamos un sencillo juego para el que creamos un pequeño mundo. La mansión del terror. Todo lo aplicado en el capítulo segundo es perfectamente válido. Si quisieramos rehacer el juego, pero esta vez de forma gráfica y manejando un personaje que se mueve a traves de las habitaciones, tendríamos que modificar un poco las estructuras de datos. En aquel juego, la descripción de las estancias de la mansión se presentaba al jugador en forma de texto. Ahora, si queremos representar gráficamente las habitaciones, hemos de conocer la situación de los elementos que la componen. Deberíamos añadir a la estructura de datos un array de los elementos que componen la habitación. Tendría el aspecto siguiente:

```
// Estructura de datos que describe un
// elemento de una habitación
struct item {
   SDL_Surface *img;
   int posx;
   int posy;
}

// Estructura de datos para las habitaciónes
struct habitacion {
   struct item elemnto[MAX_ELEMENTS];
   int norte;
   int sur;
   int este;
   int oeste;
}
```

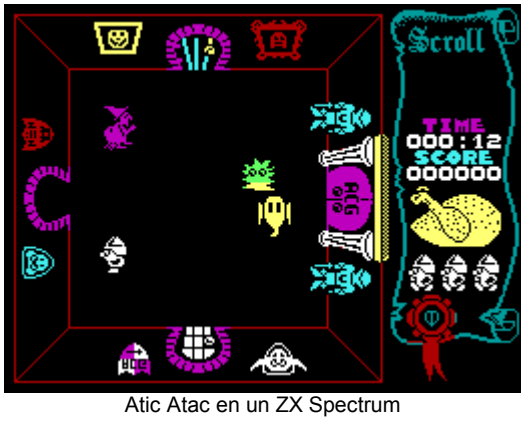
La estructura `item` describe un elemento individual de la habitación, como una cama, una mesa o una silla. Tiene tres elementos en la estructura que la describen. Por un lado sus dos coordenadas dentro de la habitación, y por supuesto, un gráfico con el que representar el objeto. En la estructura `habitacion`, sólo hemos hecho un cambio. Hemos sustituido el campo que contenía el texto descriptor de la habitación por un array de elementos de tipo `item`. Un posible código que mostrara los elementos contenidos en una habitación podría tener el siguiente aspecto.

```
SDL_Rect dest;
struct habitacion room[NUM_ROOMS];

// Mostrar elementos de la habitación
for (int i=1 ; i<=MAX_ELEMENTS ; i++) {
   dest.x = room[hab_actual].elemento[i].posx;
    dest.y = room[hab_actual].elemento[i].posx;
    SDL_BlitSurface(room[hab_actual].elemento[i].img, NULL, screen,
&dest);
}
```

Como ves, la elección de una buena estructura de datos es fundamental a la hora de hacer un buen diseño. Creeme, puede ser la diferencia entre un código limpio y elegante o un spaghetti-code (código mal estructurado). Y todo esto sin contar con los doleres de cabeza que tendras que sufrir para hacerlo funcionar. Hacer buenas estructuras de datos es algo parecido a un arte y se mejora con la experiencia. Sólo un consejo: Manten las cosas simples. La estructura de datos tiene que permitir que el código que hace uso de ella sea lo más sencillo y simple posible.

Este primer intento de crear un mundo visual parece bueno. Algunos antiguos juegos como *Atic Atac* han utilizado técnicas similares a estas.



Atic Atac en un ZX Spectrum

El problema es que probablemente se consigue poco detalle visual. En nuestro juego vamos a utilizar una técnica distinta basada en *tiles*. En español tile significa baldosa o azulejo. Esto nos da una idea de en qué consiste la técnica. La idea es construir la imagen a mostrar en la pantalla mediante tiles de forma cuadrada, como si enlosaramos una pared. Mediante tiles distintos podemos formar cualquier imagen. La siguiente figura pertenece al juego Uridium, un juego de naves o shooter de los años 80 parecido al que vamos a desarrollar.

Uridium

Las líneas rojas dividen los tiles empleados para construir la imagen.

Almacenando y mostrando tiles

En nuestro juego vamos a manejar mapas sencillos. Van a estar compuestos por mosaicos de tiles simples. Algunos juegos tienen varios niveles de tiles (llamados capas). Por ahora, vamos a almacenar la información sobre nuestro mapa en un array de enteros tal como este.

```
int mapa[100] = {  0,0,0,0,0,0,0,0,0,0,
                   0,0,0,0,0,2,0,0,0,0,
                   0,0,1,0,0,0,0,1,0,0,
                   2,0,0,0,0,0,0,0,0,0,
                   0,0,0,0,1,0,0,0,2,0,
                   0,0,0,0,0,0,0,0,0,0,
                   0,2,0,0,0,0,0,0,0,0,
                   0,0,0,0,0,0,0,1,0,0,
                   0,0,1,0,0,0,0,0,0,0,
                   0,0,0,0,0,0,0,0,0,0};
```

Este array representa un array de 10x10 tiles. Vamos a utilizar los siguientes tiles, cada uno con un tamaño de 64x64 píxeles.


Tiles utilizados en nuestro juego

Para cargar y manejar los tiles vamos a apoyarnos en la librería de manejo de sprites que desarrollamos en el capítulo anterior.

```
CFrame tile1;
CFrame tile2;
CFrame tile3;
```

```
CSprite suelo[3];

tile1.load("tile0.bmp");
suelo[0].addframe(tile1);

tile2.load("tile1.bmp");
suelo[1].addframe(tile2);

tile3.load("tile2.bmp");
suelo[2].addframe(tile3);
```

Hemos creado un array de tres sprites, uno por cada tile que vamos a cargar.

El proceso de representación del escenario consiste en ir leyendo el mapa y dibujar el sprite leido en la posición correspondiente. El siguiente código realiza este proceso.

```
int i,j,x,y,t;

//dibujar escenario
for (i=0 ; i<10 ; i++) {
  for (j=0 ; j<10 ; j++) {
    t=mapa[i*10+j];
    // calculo de la posición del tile
    x=j*64;
    y=(i-1)*64;

    // dibujamos el tile
    suelo[t].setx(x);
    suelo[t].sety(y);
    suelo[t].draw(screen);
  }
}
```

El mapa es de 10x10, así que los dos primero bucles se encargan de recorrer los tiles. La variable `t`, almacena el valor del tile en cada momento. El calculo de la coordenada de la pantalla en la que debemos dibujar el tile tampoco es complicada. Al tener cada tile 64x64 píxeles, sólo hay que multiplicar por 64 el valor de los contadores `i` o `j`, correspondiente a los bucles, para obtener la coordenada de pantalla. Seguidamente se muestra el código completo del ejemplo6_1.

```
/*****************************************************************************
 Ejemplo6_1
 (C) 2003 by Alberto Garcia Serrano
 Programación de videojuegos con SDL
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <SDL.h>
#include "csprite.h"


// Mapa de tiles
// cada tile tiene 64x64 píxeles
// una pantalla tiene 10x10 tiles
int mapa[100]= {0,0,0,0,0,0,0,0,0,0,
            0,0,0,0,0,2,0,0,0,0,
            0,0,1,0,0,0,0,1,0,0,
            2,0,0,0,0,0,0,0,0,0,
            0,0,0,0,1,0,0,0,2,0,
            0,0,0,0,0,0,0,0,0,0,
            0,2,0,0,0,0,0,0,0,0,
            0,0,0,0,0,0,0,1,0,0,
            0,0,1,0,0,0,0,0,0,0,
            0,0,0,0,0,0,0,0,0,0};

SDL Surface *screen;
CFrame fnave;
CFrame fmalo;
CFrame tile1;
CFrame tile2;
CFrame tile3;
CSprite nave(1);
CSprite malo(1);
CSprite suelo[3];
SDL_Rect rectangulo;
SDL Joystick *joystick;
int joyx, joyy;
int done=0;

// estructura que contiene la información
// de nuestro avión
struct minave {
   int x,y;
} jugador;

// Estructura que contiene información
// del avión enemigo
struct naveenemiga {
   int x,y;
} enemigo;


// Dibuja los esprites en la pantalla
void DrawScene(SDL Surface *screen) {
 int i,j,x,y,t;

   //dibujar escenario
   for (i=0 ; i<10 ; i++) {
      for (j=0 ; j<10 ; j++) {
         t=mapa[i*10+j];
         // calculo de la posición del tile
         x=j*64;
         y=(i-1)*64;

         // dibujamos el tile
         suelo[t].setx(x);
         suelo[t].sety(y);
         suelo[t].draw(screen);
      }
   }

   // dibuja avión
   nave.setx(jugador.x);
   nave.sety(jugador.y);
   nave.draw(screen);
```

```
   // Dibuja enemigo
   malo.setx(enemigo.x);
   malo.sety(enemigo.y);
   malo.draw(screen);

   // ¿ha colisionado con la nave?
   if (malo.colision(nave) == TRUE) {
      done=1;
   }

   // Mostramos todo el frame
   SDL_Flip(screen);
}


// Inicializamos estados
void inicializa() {
   jugador.x=300;
   jugador.y=300;
   enemigo.x=100;
   enemigo.y=100;
}


void finaliza() {

   // finalizamos los sprites
   nave.finalize();
   malo.finalize();
   suelo[0].finalize();
   suelo[1].finalize();
   suelo[2].finalize();

   // cerramos el joystick
   if (SDL_JoystickOpened(0)) {
      SDL_JoystickClose(joystick);
   }
}


// Preparamos los esprites
int InitSprites() {

   fnave.load("minave.bmp");
   nave.addframe(fnave);

   fmalo.load("nave.bmp");
   malo.addframe(fmalo);

   tile1.load("tile0.bmp");
   suelo[0].addframe(tile1);

   tile2.load("tile1.bmp");
   suelo[1].addframe(tile2);

   tile3.load("tile2.bmp");
   suelo[2].addframe(tile3);

   return 0;
}



int main(int argc, char *argv[]) {

SDL Event event;
Uint8 *keys;


   if (SDL_Init(SDL_INIT_VIDEO|SDL_INIT_JOYSTICK) < 0) {
      printf("No se pudo iniciar SDL: %s\n",SDL_GetError());
      return 1;
   }

   screen = SDL_SetVideoMode(640,480,24,SDL_HWSURFACE);
   if (screen == NULL) {
      printf("No se puede inicializar el modo gráfico: \n",SDL_GetError());
```

```
      return 1;
   }

   atexit(SDL_Quit);

   inicializa();

   InitSprites();

   while (done == 0) {

      // dibujamos el frame
      DrawScene(screen);

      // consultamos el estado del teclado
      keys=SDL_GetKeyState(NULL);

      // consultamos el estado del joystick
      SDL_JoystickUpdate();

      joyx = SDL_JoystickGetAxis(joystick, 0);
      joyy = SDL_JoystickGetAxis(joystick, 1);

      if ((keys[SDLK_UP] || joyy < -10) && (jugador.y > 0)) {jugador.y=jugador.y-(5);}
      if ((keys[SDLK_DOWN] || joyy > 10) && (jugador.y < 460)) {jugador.y=jugador.y+(5);}
      if ((keys[SDLK_LEFT] || joyx < -10) && (jugador.x > 0)) {jugador.x=jugador.x-(5);}
      if ((keys[SDLK_RIGHT] || joyx > 10) && (jugador.x < 620)) {jugador.x=jugador.x+(5);}


      while (SDL_PollEvent(&event)) {

         if (event.type == SDL_QUIT) {done=1;}

         if (event.type == SDL_KEYDOWN || event.type == SDL_JOYBUTTONDOWN) {

            if (event.key.keysym.sym == SDLK_ESCAPE) {
               done=1;
            }

         }

      }
   }


   finaliza();

   return 0;
}
```

El resultado de ejecutar este código de ejemplo se muestra en la próxima figura. El programa hace exactamente lo mismo que el del anterior capítulo, es decir, podrás mover el avión por la pantalla hasta colisionar con el avión enemigo. La diferencia es que ahora tenemos un bonito fondo en lugar de aquel color negro.

Nota que hemos eliminado el código necesario para borrar la anterior posición del avión en su movimiento. Al redibujar todo el fondo cada vez, ya no es necesario, ya que se actualiza la pantalla completa.

Diseñando el mapa

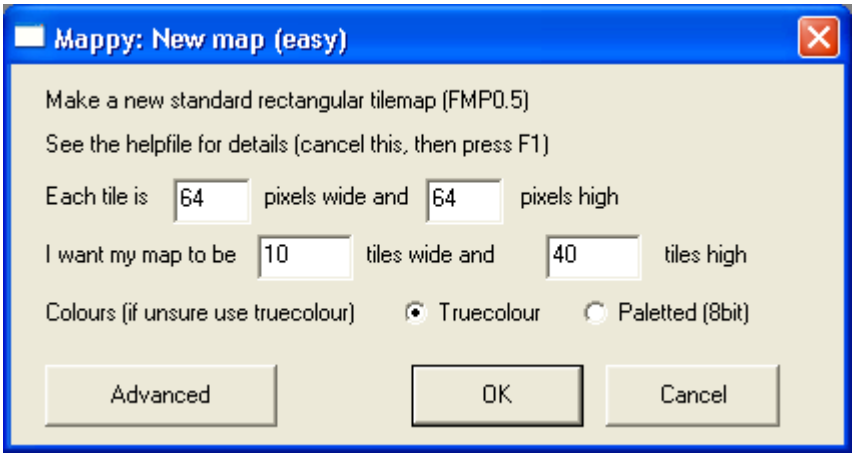
El código anterior es válido siempre que no se den dos condiciones. La primera es que el mapa no ocupe la pantalla, y la segunda es que el mapa no sea demasiado complejo. Imagina tener que hacer inmensos y complejos arrays de enteros para representa un gran mapa.

Habitualmente, los programadores de videojuegos usan lo que se llaman *editores de mapas*. Un editor de mapas nos permite diseñar nuestro mapa de forma visual. Así vamos viendo como va quedando mientras lo diseñamos. Hay dos opciones, o desarrollar uno propio (no es demasiado complejo, aunque lo parezca) o utilizar alguno existente. Por ahora optaremos por la segunda opción. Hay algunos buenos programas editores de mapa, pero nosotros vamos a utilizar en este libro uno llamado Mappy, que es sencillo, flexible, y sobre todo, gratuito. Puedes descargarlo de la siguiente dirección web: *http://www.tilemap.co.uk/mappy.php* .

Este programa permite configurar el formato del fichero que genera, con lo que nos sirve perfectamente. El formato que vamos a utilizar no puede ser más sencillo. Es un fichero binario (ojo, no de texto) en el que cada byte representa una posición consecutiva del mapa (un elemento del array en el anterior programa). Nuestro mapa va a tener 10x40 tiles, es decir, 10 tiles en horizontal y 40 en vertical. Por lo tanto, si cada tile está representado por un byte, el archivo para el mapa ocupará exactamente 400 bytes. Para que Mappy pueda generar este tipo de archivo hay que configurarlo editando en archivo MAPWIN.INI. Más concretamente hay que cambiar la línea **maptype="LW4H4A4-1"** por ésta otra **maptype="LA1-1"**. No voy a

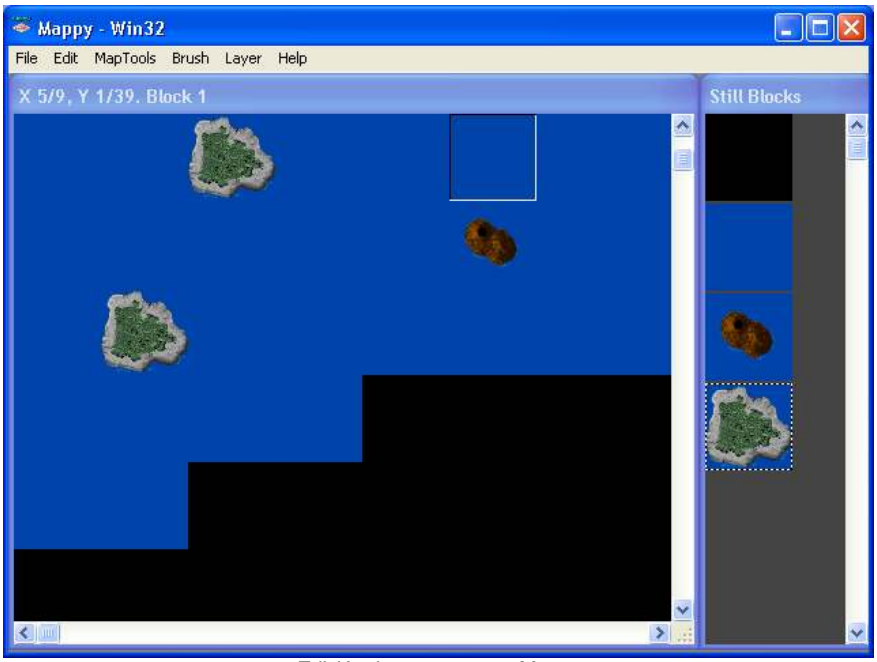
entrar en detalle sobre este fichero de configuración, pero si tienes curiosidad puedes consultar el manual de Mappy.

Veamos paso a paso como creamos un mapa para nuestro juego. Ejecuta el programa y seleciona *New Map...* del menu *file*. Aparecerá un diálogo preguntandote si quieres utilizar el diálogo avanzado o el sencillo. Pulsa *No* para usar el sencillo.



Introduce los valores que ves en la figura, es decir, 64 pixels tanto de anchura como de altura para los tiles. En la anchura de tiles ponemos 10 y en la altura 40. Seguidamente pulsa *OK*.

Ahora vamos a cargar los tiles. Lo mejor es nombrar los tiles como tile1.bmp, tile2.bmp, tileN.bmp, etc... Así luego no tendremos problema con el orden de los tiles. Para cargar los tiles en el programa seleccionamos *import...* del menu *file*. Se nos abrirá un diálogo para seleccionar un archivo. Cargamos el primer tile. Nos aparecerá un dialogo que reza: *"Make all imported graphics into NEW block structures?"*. Respondemos que sí y repetimos este proceso para todos los tiles del juego. Los tiles que se van cargando se van situando en la ventana del laterar derecho. A partir de ahí sólo tienes que seleccionar el tile que quieres dibujar y situarlo en la ventana central en el lugar deseado.



Edición de un mapa con Mappy

Una vez que hemos terminado el diseño del mapa, hemos de grabarlo selecionando *save as...* del menú file. Podemos grabar el archivo con dos extensiones diferentes: .FMP y .MAP. El formato .FMP es el formato interno de Mappy, y el formato .MAP es nuestro formato (el que definimos al editar el archivo MAPWIN.INI). Si lo grabamos exclusivamente en nuestro formato, es decir .MAP, Mappy no será capaz de volver a leerlo, así que lo mejor es guardarlo en formato .FMP y seguidamente volver a guardarlo en formato .MAP.
Cargar el mapa desde nuestro juego es sumamente sencillo. Las siguientes líneas realizan esta tarea.

```
#define MAXMAP 400
char mapa[401];
FILE *f;

// Carga del mapa
if((f=fopen("map.map","r")) != NULL) {
   c=fread(mapa,MAXMAP,1,f);
   fclose(f);
}
```

El array `mapa`, contiene ahora toda la información del mapa.

Scrolling

Se hace evidente que un mapa de 10x40 tiles no cabe en la pantalla. Lo lógico es que según se mueva nuestro avión por el escenario, el mapa avance en la misma dirección. Este desplazamiento del escenario se llama *scrolling*. Si nuestro avión avanza un tile en la pantalla, hemos de dibujar el escenario, pero desplazado (offset) un tile. Desafortunamente, haciendo esto exclusivamente, veríamos como el escenario va dando saltitos, y lo que buscamos es un scroll suave. El siguiente fragmento de código cumple este cometido.

```
#define MAXMAP 400
indice=MAXMAP-100;
indice_in=0;

// movimiento del scenario (scroll)
indice_in+=2;
if (indice_in>=64) {
   indice_in=0;
   indice-=10;
}

if (indice <= 0) {
   indice=MAXMAP-100; // si llegamos al final, empezamos de nuevo.
   indice_in=0;
}

//dibujar escenario
for (i=0 ; i<10 ; i++) {
   for (j=0 ; j<10 ; j++) {
      t=mapa[indice+(i*10+j)];
      // calculo de la posición del tile
      x=j*64;
      y=(i-1)*64+indice_in;

      // dibujamos el tile
      suelo[t].setx(x);
```

```
      suelo[t].sety(y);
      suelo[t].draw(screen);
   }
}
```

Este código está basado en el ejemplo6_1. Como diferencia encontramos dos nuevas variables. La variable `indice` contiene el desplazamiento (en bytes) a partir del cual se comienza a dibujar el mapa. La variable `indice_in`, es la encargada de realizar el scroll fino. Al dibujar el tile, se le suma a la coordenada `Y` el valor de la variable `indice_in`, que va aumentando en cada iteración. Cuando esta variable alcanza el valor 64, es decir, la altura del tile, ponemos la variable a 0 y restamos 10 a la variable `indice`, o lo que es lo mismo, el offset a partir del que dibujamos el mapa. Se realiza una resta porque la lectura del mapa la hacemos de abajo a arriba (del último byte al primero). Recuerda que el mapa tiene 10 tiles de anchura. Es por eso que restamos 10 a la variable `indice`. Una vez que llegamos al principio del mapa, comenzamos de nuevo por el final, de forma que se va repitiendo el mismo mapa de forma indefinida. Ten en cuenta que las primeras 10 filas del mapa tienen que ser iguales que las 10 últimas si no quieres notar un molesto salto cuando recomienza el recorrido del mapa. En el ejemplo6_2 vemos el código completo.

```
/*****************************************************************************
 Ejemplo6_2
 (C) 2003 by Alberto Garcia Serrano
 Programación de videojuegos con SDL
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <SDL.h>
#include "csprite.h"

#define MAXMAP 400

SDL Surface *screen;
CFrame fnave;
CFrame fmalo;
CFrame tile1;
CFrame tile2;
CFrame tile3;
CSprite nave(1);
CSprite malo(1);
CSprite suelo[3];
SDL_Rect rectangulo;
SDL_Joystick *joystick;
char mapa[401];
int joyx, joyy;
int done=0;
int indice, indice_in;
FILE *f;


// estructura que contiene la información
// de nuestro avión
struct minave {
   int x,y;
} jugador;


// Estructura que contiene información
// del avión enemigo
struct naveenemiga {
   int x,y;
} enemigo;


// Dibuja el escenario
void DrawScene(SDL_Surface *screen) {
 int i,j,x,y,t;


   // movimiento del scenario (scroll)
   indice_in+=2;
   if (indice_in>=64) {
      indice_in=0;
      indice-=10;
   }

   if (indice <= 0) {
      indice=MAXMAP-100; // si llegamos al final, empezamos de nuevo.
      indice in=0;
   }

   //dibujar escenario
   for (i=0 ; i<10 ; i++) {
      for (j=0 ; j<10 ; j++) {
         t=mapa[indice+(i*10+j)];
         // calculo de la posición del tile
         x=j*64;
         y=(i-1)*64+indice_in;

         // dibujamos el tile
         suelo[t].setx(x);
         suelo[t].sety(y);
         suelo[t].draw(screen);
      }
   }
```

```
   // dibuja avión
   nave.setx(jugador.x);
   nave.sety(jugador.y);
   nave.draw(screen);


   // Dibuja enemigo
   malo.setx(enemigo.x);
   malo.sety(enemigo.y);
   malo.draw(screen);

   // ¿ha colisionado con la nave?
   if (malo.colision(nave) == TRUE) {
      done=1;
   }

   // Mostramos todo el frame
   SDL_Flip(screen);
}


// Inicializamos estados
void inicializa() {
 int c;

   jugador.x=300;
   jugador.y=300;
   enemigo.x=100;
   enemigo.y=100;

   indice=MAXMAP-100;
   indice in=0;


   // Carga del mapa
   if((f=fopen("map.map","r")) != NULL) {
      c=fread(mapa,MAXMAP,1,f);
      fclose(f);
   }

}


void finaliza() {

   // finalizamos los sprites
   nave.finalize();
   malo.finalize();
   suelo[0].finalize();
   suelo[1].finalize();
   suelo[2].finalize();

   // cerramos el joystick
   if (SDL_JoystickOpened(0)) {
      SDL JoystickClose(joystick);
   }
}


// Preparamos los esprites
int InitSprites() {

   fnave.load("minave.bmp");
   nave.addframe(fnave);

   fmalo.load("nave.bmp");
   malo.addframe(fmalo);

   tile1.load("tile0.bmp");
   suelo[0].addframe(tile1);

   tile2.load("tile1.bmp");
   suelo[1].addframe(tile2);

   tile3.load("tile2.bmp");
   suelo[2].addframe(tile3);
```

```
   return 0;
}


int main(int argc, char *argv[]) {

SDL_Event event;
Uint8 *keys;


   if (SDL_Init(SDL_INIT_VIDEO|SDL_INIT_JOYSTICK) < 0) {
      printf("No se pudo iniciar SDL: %s\n",SDL_GetError());
      return 1;
   }

   screen = SDL_SetVideoMode(640,480,24,SDL_HWSURFACE);
   if (screen == NULL) {
      printf("No se puede inicializar el modo gráfico: \n",SDL_GetError());
      return 1;
   }

   atexit(SDL_Quit);

   inicializa();

   InitSprites();

   while (done == 0) {

      // dibujamos el frame
      DrawScene(screen);

      // consultamos el estado del teclado
      keys=SDL_GetKeyState(NULL);

      // consultamos el estado del joystick
      SDL JoystickUpdate();

      joyx = SDL_JoystickGetAxis(joystick, 0);
      joyy = SDL_JoystickGetAxis(joystick, 1);

      if ((keys[SDLK_UP] || joyy < -10) && (jugador.y > 0)) {jugador.y=jugador.y-(5);}
      if ((keys[SDLK DOWN] || joyy > 10) && (jugador.y < 460)) {jugador.y=jugador.y+(5);}
      if ((keys[SDLK LEFT] || joyx < -10) && (jugador.x > 0)) {jugador.x=jugador.x-(5);}
      if ((keys[SDLK_RIGHT] || joyx > 10) && (jugador.x < 620)) {jugador.x=jugador.x+(5);}


      while (SDL_PollEvent(&event)) {

         if (event.type == SDL QUIT) {done=1;}

         if (event.type == SDL_KEYDOWN || event.type == SDL_JOYBUTTONDOWN) {

            if (event.key.keysym.sym == SDLK_ESCAPE) {
               done=1;
            }

         }

      }
   }


   finaliza();

   return 0;
}
```

Capítulo 7

Enemigos, disparos y explosiones

Perdona siempre a tu enemigo. No hay nada que le enfurezca más.
Oscar Wilde.

En los ejemplos del anterior capítulo, nuestro avión enemigo está inmóvil. En este capítulo vamos a hacer que nuestros enemigos cobren vida propia. Existen multiples técnicas relacionadas con la inteligencia artificial (IA) y que son ampliamente utilizadas en programación de juegos. La IA es un tópico lo suficientemente extenso como para rellenar varios libros del tamaño del que tienes ahora entre manos. Aún así, exploraremos algunas sencillas técnicas que nos permitiran dotar a los aviones enemigos de nuestro juego de una chispa vital. También vamos a hacer diaparar a los aviones enemigos y al nuestro, explosiones incluidas.

Tipos de inteligencia

Hay, al menos, tres tendencias dentro del campo de la inteligencia artificial.

- Redes neuronales
- Algoritmos de búsqueda
- Sistemas basados en conocimiento

Son tres enfoque diferentes que tratan de buscar un fin común. No hay un enfoque mejor que los demás, la elección de uno u otro depende de la aplicación.

Una red neuronal trata de simular el funcionamiento del cerebro humano. El elemento básico de una red neuronal es la neurona. En una red neuronal, un conjunto de neuronas trabajan al unísono para resolver un problema. Al igual que un niño tiene que aprender al nacer, una red de neuronas artificial tiene que ser entrenada para poder realizar su cometido. Este aprendizaje puede ser supervisado o no supervisado, dependiendo si hace falta intervención humana para entrenar a la red de neuronas. Este entrenamiento se realiza normalmente mediante ejemplos. La aplicación de las redes neuronales es efectiva en campos en los que no existen algoritmos concretos que resuelvan un problema o sean demasiado complejos de computar. Donde más se aplican es en problemas de reconocimiento de patrones y pronósticos.

El segundo enfoque es el de los algoritmos de búsqueda. Es necesario un conocimiento razonable sobre estructuras de datos como árboles y grafos. Una de las aplicaciones interesantes, sobre todo para videojuegos, es la busqueda de caminos (pathfinding). Seguramente has jugado a juegos de estrategia como *Starcraft*, *Age of Empires* y otros del estilo. Puedes observar que cuando das la orden de movimiento a uno de los pequeños personajes del juego, éste se dirige al punto indicado esquivando los obstáculos que encuantra en su camino. Este algoritmo de búsqueda en grafos es llamado A*. Tomemos como ejemplo el mapa de la mansión del capítulo 2. Suponiendo que el jugardor está en la habitación 1, éste es el arbol de los posible caminos que puede tomar para escapar de la mansión.

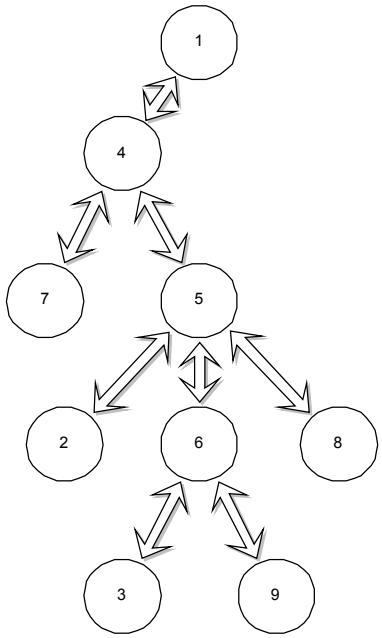


Figura 7.1. Arbol de los posibles caminos dentro de mapa de juego

Cada círculo representa un nodo del arbol. El número que encierra es el número de habitación. Si quisieramos encontrar la salida, usaríamos un algoritmo de búsqueda (como por ejemplo A*) para recorrer todos los posibles caminos y quedarnos con el que nos interesa. El objetivo es buscar el nodo 8, que es el que tiene la puerta de salida. El camino desde la habitación 1 es: 1 4 5 8. El algoritmo A* además de encontrar el nodo objetivo, nos asegura que es el camino más corto. No vamos a entrar en más detalle, ya que cae fuera de las pretensiones de este libro profundizar en la implementación de algoritmos de búsqueda.

Por último, los sistemas basados en reglas se sirven, valga la redundancia, de conjuntos de reglas y hechos. Los hechos son informaciones relativas al problema y a su universo. Las reglas son precisamente eso, reglas aplicables a los elementos del universo y que permiten llegar a deducciones simples. Veamos un ejemplo:

Hechos: Las moscas tienen alas.
Las hormigas no tienen alas.

Reglas: Si (x) tiene alas, entonces vuela.

Un sistema basado en conocimiento, con estas reglas y estos hechos es capaz de deducir dos cosas. Que las moscas vuelan y que las hormigas no.

Si (la mosca) tiene alas, entonces vuela.

Uno de los problemas de los sistemas basados en conocimiento es que pueden ocurrir situaciones como estas.

Si (la gallina) tiene alas, entonces vuela.

Desgraciadamente para las gallinas, éstas no vuelan. Puedes observar que la construcción para comprobar reglas es muy similar a la construcción IF/THEN de los lenguajes de programación.

Comportamientos y máquinas de estado

Una máquina de estados está compuesta por por una serie de estados y una serie de reglas que indican en que casos se pasa de un estado a otro. Estas máquinas de estados nos perniten modelar comportamientos en los personajes y elementos del juego. Vamos a ilustrarlo con un ejemplo. Imagina que en el juego de la mansión del terror del capítulo 2 hay un zombie. El pobre no tiene una inteligencia demasiado desarrollada y sólo es capaz de andar hasta que se pega contra la pared. Cuando sucede esto, lo único que sabe hacer es girar 45 grados a la derecha y continuar andando. Vamos a modelar el comportamiento del zombie con una máquina de estados. Para ello primero tenemos que definir los posibles estados.

- Andando (estado 1)
- Girando (estado 2)

Las reglas que hacen que el zombie cambie de un estado a otro son las siguientes.

- Si está en el estado 1 y choca con la pared pasa al estado 2.
- Si esta en el estado 2 y ha girado 45 grados pasa al estado 1.

Con estos estados y estas reglas podemos construir el grafo que representa a nuestra máquina de estados.

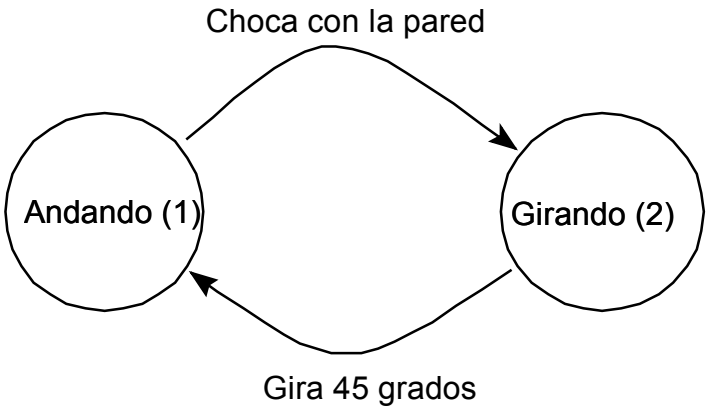


Figura 7.2.

La implementación de la máquina de estado es muy sencilla. La siguiente función simula el comportamiento del zombie.

```
int angulo;

void zombie() {
 int state, angulo_tmp;

   // estado 1
   if (state == 1) {
      andar();
      if (colision()) {
         state=2;
         angulo_tmp=45;
      }
   }

   // estado 2
   if (state == 2) {
      angulo_tmp=angulo_tmp-1;
      angulo=angulo+1;
      if (angulo_tmp <= 0) {
         state=1;
      }
   }
}
```

Éste es un ejemplo bastante sencillo, sin embargo utilizando este método podrás crear comportamientos inteligentes de cierta complejidad. En el programa ejemplo7_1 el avión enemigo va a tener un movimiento lateral, de lado a lado de la pantalla de forma indefinida. La máquina de estados es la siguiente.

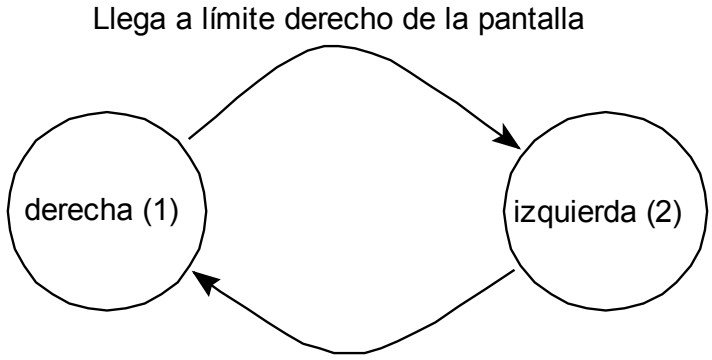


Figura 7.3.

Disparos y explosiones

Ahora que nuestro aviones enemigos son capaces de comportarse tal y como queremos que lo hagan, estamos muy cerca de poder completar nuestro juego. Lo siguiente que vamos a hacer es añadir la capacidad de disparar a nuestro avión. También vamos a ver como gestionar la explosión de la nave enemiga cuando es alcanzada. Lo primero que necesitamos es una estructura de datos para contener la información de los disparos.

```
// Estructura que contine información
// de los disparos de nuestro avión
```

```
struct disparo {
   int x,y;
} bala[MAXBALAS+1];
```

La constante `MAXBALAS` contiene el número máximo de balas simultáneas que nuestro avión puede disparar. Por lo tanto, tenemos un array de balas listas para utilizar. Cada vez que pulsamos la tecla de disparo, hemos de utilizar uno de los elementos de este array para almacenar la información del disparo, por lo tanto la primera tarea es encontrar un elemento libre (que no se esté usando) dentro del array. Una vez encontrado, hemos de inicializar el disparo con la posición del avión. La función `creadisparo()` realiza este trabajo.

```
void creadisparo() {

   int libre=-1;

   // ¿Hay alguna bala libre?
   for (int i=0 ; i<=MAXBALAS ; i++) {
      if (bala[i].x==0)
         libre=i;
   }

   // Hay una bala
   if (libre>=0) {
      bala[libre].x=nave.getx();
      bala[libre].y=nave.gety()-15;
   }
}
```

Para saber si una bala está o no libre, usaremos su coordenada X. Si vale 0 asumiremos que está libre, si no, es un elemento del array en uso y tendremos que buscar otro.

En cada vuelta del game loop, hemos de dibujar los disparos activos si es que los hubiera, el código siguiente se encarga de ello.

```
void muevebalas() {
 int i;

   for (i=0 ; i<=MAXBALAS ; i++) {

      // si la pos.X del desparo no es 0,
      // es una bala activa.
      if (bala[i].x != 0) {
         bala[i].y=bala[i].y-5;

      // si el disparo sale de la pantalla la
      // desactivamos
         if (bala[i].y < 0) {
            bala[i].x=0;
         }
      }
   }
}
```

Hemos de recorrer todo el array de balas para localizar cuáles de ellas necesitan ser actualizadas en la pantalla. Si la coordenada X del disparo es distinta de 0, la bala está activa y hay que moverla en el eje Y (sentido ascendente). Es importante comprobar si la bala sale de la pantalla para poder liberar el elemento del array que ocupa, ya que si no, pronto nos quedaríamos sin poner disparar porque no habría elementos libres en el array.

Nos resta dibujar los disparos. De nuevo, necesitamos recorrer todo el array de disparos y dibujar los que están activos.

```
// dibuja disparos
for (i=0 ; i<=MAXBALAS ; i++) {
   if (bala[i].x != 0) {
      mibala.setx(bala[i].x);
      mibala.sety(bala[i].y);
      mibala.draw(screen);
   }
}
```

La misión de los disparos es, como ya sabes, destruir el avión enemigo. Lo lógico es que el avión impactado salte por los aires y produzca una explosión. Vamos a utilizar la siguiente estructura de datos para contener la explosión.

```
// Estructura que contiene información
// de la explosión
struct explosion {
   int activo,x,y,nframe;
} exp;
```

Además de las coordenadas de la explosión, necesitamos un campo que nos indique si la explosión está activa. El sprite utilizado para la explosión tiene alguna diferencia con los utilizados hasta ahora. Este sprite está formado por los 7 frames que componen la animación de la explosión. El campo `nframe` contine el frame que se está mostrando en la actualidad, por lo tanto, en cada repintado de la imagen, hemos de incrementar `nframe` para crear el efecto de animación. Cuando esta variable llega a 7, es decir, el número de frames que componen la animación, nos indica que la explosión ha terminado, por lo que hay que desactivarla. El código siguiente realiza la tarea de dibujar la explosión.

```
// dibujar explosion
if (exp.activo==1) {
   explode.selframe(exp.nframe);
   explode.setx(exp.x);
   explode.sety(exp.y);
   explode.draw(screen);
   exp.nframe=exp.nframe+1;
   if (exp.nframe>=7) {
      exp.activo=0;
      done=1;
   }
}
```

La explosión ha de activarse cuando el sprite del avión enemigo colisione con una bala. Al producirse el impacto, hemos de eliminar el avión enemigo para que no siga dibujándose y activar la explosión. Las coordenadas de la explosión serán las mismas que las que tenía la nave enemiga.

```
// ¿ha colisionado el disparo con la nave?
if (malo.colision(mibala) == TRUE) {
   enemigo.estado=0;
   exp.activo=1;
   exp.nframe=1;
```

```
   exp.x=enemigo.x;
   exp.y=enemigo.y;
}
```

Tienes el código completo en el programa ejemplo7_1.

Figura 7.4.

```
/*************************************************************************
 Ejemplo7_1
 (C) 2003 by Alberto Garcia Serrano
 Programación de videojuegos con SDL
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <SDL.h>
#include "csprite.h"

#define MAXMAP 400
#define MAXBALAS 8

SDL Surface *screen;
CFrame fnave;
CFrame fmalo;
CFrame tile1;
CFrame tile2;
CFrame tile3;
CFrame labala;
CFrame ex1;
CFrame ex2;
CFrame ex3;
CFrame ex4;
CFrame ex5;
CFrame ex6;
CFrame ex7;
CSprite nave(1);
CSprite malo(1);
CSprite suelo[3];
CSprite mibala(1);
CSprite explode(8);
SDL_Rect rectangulo;
SDL_Joystick *joystick;
char mapa[401];
int joyx, joyy;
int done=0;
int indice, indice_in;
FILE *f;



// estructura que contiene la información
// de nuestro avión
struct minave {
   int x,y;
} jugador;


// Estructura que contiene información
// del avión enemigo
struct naveenemiga {
   int x,y,estado;
} enemigo;


// Estructura que contine información
// de los disparos de nuestro avión
struct disparo {
   int x,y;
} bala[MAXBALAS+1];


// Estructura que contiene información
// de la explosión
struct explosion {
   int activo,x,y,nframe;
} exp;




void muevenave() {

   // estado 1. Movimiento a la derecha.
   if (enemigo.estado == 1) {
```

```
      enemigo.x=enemigo.x+2;
      if (enemigo.x>600) enemigo.estado=2;
   }

   // estado2. Movimiento a la izquierda.
   if (enemigo.estado == 2) {
      enemigo.x=enemigo.x-2;
      if (enemigo.x<40) enemigo.estado=1;
   }
}


void muevebalas() {
 int i;

   for (i=0 ; i<=MAXBALAS ; i++) {

      // si la pos.X del desparo no es 0,
      // es una bala activa.
      if (bala[i].x != 0) {
         bala[i].y=bala[i].y-5;

         // si el disparo sale de la pantalla la desactivamos
         if (bala[i].y < 0) {
            bala[i].x=0;
         }
      }
   }
}


// Dibuja el escenario
void DrawScene(SDL_Surface *screen) {
 int i,j,x,y,t;


   // movimiento del scenario (scroll)
   indice in+=2;
   if (indice_in>=64) {
      indice_in=0;
      indice-=10;
   }

   if (indice <= 0) {
      indice=MAXMAP-100; // si llegamos al final, empezamos de nuevo.
      indice_in=0;
   }

   //dibujar escenario
   for (i=0 ; i<10 ; i++) {
      for (j=0 ; j<10 ; j++) {
         t=mapa[indice+(i*10+j)];
         // calculo de la posición del tile
         x=j*64;
         y=(i-1)*64+indice in;

         // dibujamos el tile
         suelo[t].setx(x);
         suelo[t].sety(y);
         suelo[t].draw(screen);
      }
   }

   // dibuja avión
   nave.setx(jugador.x);
   nave.sety(jugador.y);
   nave.draw(screen);


   // dibuja enemigo
   if (enemigo.estado != 0) {
      malo.setx(enemigo.x);
      malo.sety(enemigo.y);
      malo.draw(screen);
   }

   // dibuja disparos
```

```
   for (i=0 ; i<=MAXBALAS ; i++) {
      if (bala[i].x != 0) {
         mibala.setx(bala[i].x);
         mibala.sety(bala[i].y);
         mibala.draw(screen);
      }
   }


   // dibujar explosion
   if (exp.activo==1) {
      explode.selframe(exp.nframe);
      explode.setx(exp.x);
      explode.sety(exp.y);
      explode.draw(screen);
      exp.nframe=exp.nframe+1;
      if (exp.nframe>=7) {
         exp.activo=0;
         done=1;
      }
   }


   // ¿ha colisionado con la nave?
   if (malo.colision(nave) == TRUE) {
      done=1;
   }

   // ¿ha colisionado el disparo con la nave?
   if (malo.colision(mibala) == TRUE) {
      enemigo.estado=0;
      exp.activo=1;
      exp.nframe=1;
      exp.x=enemigo.x;
      exp.y=enemigo.y;
   }

   // Mostramos todo el frame
   SDL_Flip(screen);
}


void creadisparo() {

   int libre=-1;

   // ¿Hay alguna bala libre?
   for (int i=0 ; i<=MAXBALAS ; i++) {
      if (bala[i].x==0)
         libre=i;
   }

   // Hay una bala
   if (libre>=0) {
      bala[libre].x=nave.getx();
      bala[libre].y=nave.gety()-15;
   }
}


// Inicializamos estados
void inicializa() {
 int i,c;

   jugador.x=300;
   jugador.y=300;
   enemigo.x=100;
   enemigo.y=100;
   explode.finalize();
   indice=MAXMAP-100;
   indice_in=0;

   enemigo.estado=1;

   exp.activo=0;
```

```
   // Inicializamos el array de balas
   for (i=0 ; i<=MAXBALAS ; i++) {
      bala[i].x=0;
      bala[i].y=0;
   }

   // Carga del mapa
   if((f=fopen("map.map","r")) != NULL) {
      c=fread(mapa,MAXMAP,1,f);
      fclose(f);
   }

}


void finaliza() {

   // finalizamos los sprites
   nave.finalize();
   malo.finalize();
   mibala.finalize();

   suelo[0].finalize();
   suelo[1].finalize();
   suelo[2].finalize();

   // cerramos el joystick
   if (SDL_JoystickOpened(0)) {
      SDL_JoystickClose(joystick);
   }
}


// Preparamos los esprites
int InitSprites() {

   fnave.load("minave.bmp");
   nave.addframe(fnave);

   fmalo.load("nave.bmp");
   malo.addframe(fmalo);

   tile1.load("tile0.bmp");
   suelo[0].addframe(tile1);

   tile2.load("tile1.bmp");
   suelo[1].addframe(tile2);

   tile3.load("tile2.bmp");
   suelo[2].addframe(tile3);

   labala.load("balas.bmp");
   mibala.addframe(labala);

   ex1.load("explode1.bmp");
   explode.addframe(ex1);

   ex2.load("explode2.bmp");
   explode.addframe(ex2);

   ex3.load("explode3.bmp");
   explode.addframe(ex3);

   ex4.load("explode4.bmp");
   explode.addframe(ex4);

   ex5.load("explode5.bmp");
   explode.addframe(ex5);

   ex6.load("explode6.bmp");
   explode.addframe(ex6);

   ex7.load("explode7.bmp");
   explode.addframe(ex7);

   return 0;
}
```

```
int main(int argc, char *argv[]) {

SDL_Event event;
Uint8 *keys;


   if (SDL Init(SDL INIT VIDEO|SDL INIT JOYSTICK) < 0) {
      printf("No se pudo iniciar SDL: %s\n",SDL_GetError());
      return 1;
   }

   screen = SDL SetVideoMode(640,480,24,SDL HWSURFACE);
   if (screen == NULL) {
      printf("No se puede inicializar el modo gráfico: \n",SDL_GetError());
      return 1;
   }

   atexit(SDL Quit);

   inicializa();

   InitSprites();

   while (done == 0) {

      // movemos el avión enemiga
      muevenave();

      // movemos los disparos
      muevebalas();

      // dibujamos el frame
      DrawScene(screen);

      // consultamos el estado del teclado
      keys=SDL GetKeyState(NULL);

      // consultamos el estado del joystick
      SDL_JoystickUpdate();

      joyx = SDL JoystickGetAxis(joystick, 0);
      joyy = SDL JoystickGetAxis(joystick, 1);

      if ((keys[SDLK_UP] || joyy < -10) && (jugador.y > 0)) {jugador.y=jugador.y-(5);}
      if ((keys[SDLK_DOWN] || joyy > 10) && (jugador.y < 460)) {jugador.y=jugador.y+(5);}
      if ((keys[SDLK_LEFT] || joyx < -10) && (jugador.x > 0)) {jugador.x=jugador.x-(5);}
      if ((keys[SDLK RIGHT] || joyx > 10) && (jugador.x < 620)) {jugador.x=jugador.x+(5);}
      if (keys[SDLK LSHIFT]) {creadisparo();}


      while (SDL_PollEvent(&event)) {

         if (event.type == SDL QUIT) {done=1;}

         if (event.type == SDL_KEYDOWN || event.type == SDL_JOYBUTTONDOWN) {

            if (event.key.keysym.sym == SDLK_ESCAPE) {
               done=1;
            }

         }

      }
   }


   finaliza();

   return 0;
}
```

Capítulo 8

¡Que comience el juego!

Todo comienzo tiene su encanto.
Goethe.

A estas alturas estás casi preparado para comenzar a hacer tu propio juego. Seguro que ya tienes alguna idea en mente, pero antes, vamos a finalizar el que tenemos entre manos. Todavía nos quedan por ver algunos detalles, que aún siendo pequeños, no dejan de ser importantes. Es este capítulo vamos a completar todos los aspectos del juego y a ofrecer su listado completo. Por cierto, aún no le hemos puesto un nombre. Lo llamaremos *1945*, como homenaje al juego 1942.

Enemigos

Nuestro juego va a tener tres tipos de aviones enemigos. El primer tipo son los kamikaze, que se lanzarán en diagonal por la pantalla en dirección de nuestro avión tratando de impactar con nosotros. El segundo tipo, los cazas, realizarán una maniobra de aproximación, y cuando nos tengan a tiro, lanzarán un proyectil para después comenzar una acción evasiva. El tercer tipo es el avión jefe, con el que tendremos que enfrentarnos al final de cada nivel, y al que necesitaremos disparar 100 veces para poder destruirlo. Además, varios de estos enemigos se encontrarán a la vez en pantalla. Para manejar los aviones enemigos necesitamos mejorar la estructura de datos que utilizamos en el capítulo anterior.

```
struct naveenemiga {
   int activo,x,y,dx,dy,tipo,estado,impactos,nframe;
} enemigo[7];
```

Como podrá haber un máximo de 7 enemigos a la vez en la pantalla, creamos un array de 7 elementos. El campo `activo`, nos indicará si se está utilizando ese elemento del array actualmente o por el contrario está libre para poder utilizarlo (igual que hacíamos con el array de disparos). Los campos `X` e `Y` son las coordenadas del avión. Los campos `DX` y `DY` son importantes: indican la dirección del avión para su movimiento. Estos valores se suman a las coordenadas del avión en cada frame. Veámoslo mejor con un ejemplo. Si queremos que nuestra nave se mueva de arriba a abajo de forma vertical, a una velocidad de 3 píxeles por frame, daremos los siguientes valores a estos campos:

DX = 0
DY = 3

La operación que se realiza en cada frame es:

X = X + DX
Y = Y + DY

Con lo que las coordenadas del avión quedarán actualizadas a la nueva posición. Dándole un valor de –3 a DY conseguiremos que el avión ascienda en vez de descender por la pantalla, y dándole un valor de 3 tanto a DX como a DY conseguiremos que se mueva de forma diagonal descendente. El campo `tipo` nos indican que tipo de enemigo es. Si vale 0 será un caza, si vale 1 será un kamikaze y si vale 3 será el avión jefe. El campo `estado` nos indica el estado del avión. Lo utilizaremos en el caza para saber si su estado es el de acercarse para disparar o el de dar la vuelta para huir. El campo `impactos` indica el número de impactos que ha sufrido el avión. Lo utilizaremos con el avión jefe al que sólo podremos destruir después de 100 impactos. Por último, el campo `nframe` lo utilizaremos para almacenar el frame actual dentro de la animación. En concreto, lo utilizaremos en el caza para realizar la animación de looping que realiza para huir después de soltar el proyectil.

La función siguiente será la encargada de ir creando enemigos.

```
void creaenemigo() {
  int libre=-1;

  // ¿Hay algún enemigo libre?
  for (int i=0 ; i<=nmalos ; i++) {
    if (enemigo[i].activo==0)
      libre=i;
  }

  // Hay un enemigo dispuesto
  if (libre>=0) {
    enemigo[libre].activo=1;
    enemigo[libre].nframe=0;
    enemigo[libre].x=rand();
    if (enemigo[libre].x > 640)
      enemigo[libre].x=(int)enemigo[libre].x % 640;
    enemigo[libre].tipo=rand();
    if (enemigo[libre].tipo >= 2)
      enemigo[libre].tipo=(int)enemigo[libre].tipo % 2; // 2 tipos de
enemigos (0,1)

    if (enemigo[libre].tipo==0) {
      enemigo[libre].y=-30;
      enemigo[libre].dx=0;
      enemigo[libre].dy=5;
      enemigo[libre].estado=0;
    }

    if (enemigo[libre].tipo==1) {
      enemigo[libre].y=-30;
      if (enemigo[libre].x>nave.getx()) {
        enemigo[libre].dx=-3;
      } else {
        enemigo[libre].dx=3;
      }
      enemigo[libre].dy=5;
```

```
        enemigo[libre].estado=0;
      }

   }
}
```

Al igual que hacíamos con los disparos, al encontrar un elemento libre del array de enemigos, lo ponemos activo e inicializamos su estado. La coordenada X inicial del avión se calcula de forma aleatoria. Tambien establecemos de forma aleatoria que tipo de enemigo es (tipo 1, es decir, el caza o 2, que es el kamikaze). Dependiendo del tipo de enemigo, inicializamos sus coordenadas (X e Y) y sus parámetros de movimiento (DX y DY). La función de movimiento sólo ha de actualizar las coordenadas del avión de la siguiente forma.

```
enemigo[i].x=enemigo[i].x+enemigo[i].dx;
enemigo[i].y=enemigo[i].y+enemigo[i].dy;
```

Durante el juego, vamos a generar un avión enemigo cada 20 ciclos de juego (o lo que es lo mismo, 20 vueltas del game loop). Para saber cuando tenemos que generar un enemigo utlizamos la fórmula ciclos%20. Para crear un jefe, en cambio, esperamos 5000 ciclos (ciclos%5000).

En lo referente a los disparos enemigos, la técnica es exacta a la utilizada en el anterior capítulo con dos diferencias. La primera es que usamos un array de disparos, igual que hacemos con los enemigos, y la otra es que introducimos en la estructura de datos un campo llamado `time`. La función de este campo es controlar el tiempo para saber cuando hacer parpadear el disparo. El disparo enemigo parpadea cada 5 ciclos para hacerlo más visible.

Niveles

Cualquier juego que se precie tiene que tener niveles. El avión jefe es el guardián del siguiente nivel. Para crear los niveles, vamos a seguir unas simples normas. Los mapas y demás archivos necesarios para los niveles van a estar almacenados en una estructura de directorios como sigue:

./levels/
  /level1
  /level2
  /levelN

Es decir, dentro del directorio level crearemos un subdirectorio por cada nivel. Dentro de cada uno de estos subdirectorios almacenaremos un archivo llamado *map.map* con la descripción del mapa. Otro archivo llamado level.dsc que contendrá en formato texto el número de tiles que compone el mapa menos 1, es decir, si tiene 7 tiles, este archivo contendrá el valor 6. Por último, estarán aquí también todos los tiles que componen el mapa nombrados como tile0.bmp, tile1.bmp, etc... El siguiente código se encarga de cargar un nivel.

```
void LoadLevel(int l) {

   InitEstados();

   leveltime=100;
```

```
  // Cargar mapa (10x40)
  char tmplevel[50];
  int i, c, ntiles;

  sprintf(tmplevel,"levels/level%d/map.map",l);
  if((f=fopen(tmplevel,"r")) != NULL) {
    c=fread(mapa,MAXMAP,1,f);
    fclose(f);
  }

  // Cargar tiles del suelo
  sprintf(tmplevel,"levels/level%d/level.dsc",l);
  if((f=fopen(tmplevel,"r")) != NULL) {
    fgets(tmplevel,255,f);
    ntiles=atoi(tmplevel);
    fclose(f);
  }

  for (i=0 ; i<=ntiles ; i++) {
    sprintf(tmplevel,"levels/level%d/tile%d.bmp",l,i);
    frsuelo[i].load(tmplevel);
    suelo[i].addframe(frsuelo[i]);
  }
}
```

A esta función le pasamos como parámetro el nivel que queremos cargar. Ella se encarga de carga el mapa en memoria y de cargar los tiles (como sprites). Al comenzar a jugar en un nivel, nos aparecerá superpuesto en la pantalla el nivel que estamos jugando de forma parpadeante durante algunos segundos. La variable `leveltime` indica el tiempo en ciclos que se mostrará el nivel. Este es el código que realiza este trabajo.

```
// Muestra el nivel en el que estamos jugando
if ((leveltime>=1) && leveltime--%2) {
  sprintf(msg,"Level %d",level);
  ttext = TTF_RenderText_Shaded(fuente,msg,fgcolor,bgcolor);
  rectangulo.y=240;
  rectangulo.x=310;
  rectangulo.w=ttext->w;
  rectangulo.h=ttext->h;
  SDL_SetColorKey(ttext,SDL_SRCCOLORKEY|SDL_RLEACCEL, SDL_MapRGB(ttext-
>format,255,0,0));
  SDL_BlitSurface(ttext,NULL,screen,&rectangulo);
}
```

Cada 2 ciclos mostramos (leveltime--%2) el mensaje, de forma que dará la apariencia de paradeo.

Temporización

Controlar la velocidad del juego es importante. No podemos permitirnos que nuestro juego se ejecute a una velocidad diferente en dos máquinas distintas. Vamos a servirnos de dos funciones para realizar este control.

```
void ResetTimeBase() {
   ini_milisegundos=SDL_GetTicks();
}

int CurrentTime() {
   fin_milisegundos=SDL_GetTicks();
   return fin_milisegundos-ini_milisegundos;
}
```

La función `ResetTimeBase()` utiliza la función `SDL_GetTicks` para almacenar el tiempo en milisegundos transcurridos desde la inicialización de SDL. Este valor, es utilizado por la función `CurrentTime()` para calcular cuantos milisegundos han transcurrido desde la llamada a `ResetTimeBase()`. El procedimiento de control es el siguiente. Al principio del game loop realizamos una llamada a ResetTimeBase() para inicializar el contador. Al finalizar el game loop utilizamos el siguiente código.

```
do {
   frametime=CurrentTime();
} while (frametime<30);
```

Este bucle se ejecutará mientras no hayan transcurrido, al menos, 30 milisegundos. Es decir, nos aseguramos de que no se van a producir más de un frame cada 30ms. De esta forma, si ejecutamos nuestro juego en un ordenador muy rápido, la velocidad se mantendrá constante e igual a la de otro ordenador más lento.

Pantalla inicial, puntación y vidas

Al jugador hay que ofrecerle información de, al menos, dos datos: las vidas que le restan y la puntuación que ha conseguido hasta al momento. La puntuación se almacena en una variable global llamada `score`. El código que la muestra en pantalla es el siguiente.

```
// dibuja la puntuación
sprintf(msg,"%d",score);
ttext = TTF_RenderText_Shaded(fuente,msg,fgcolor,bgcolor);
rectangulo.y=5;
rectangulo.x=530;
rectangulo.w=ttext->w;
rectangulo.h=ttext->h;
SDL_SetColorKey(ttext,SDL_SRCCOLORKEY|SDL_RLEACCEL,     SDL_MapRGB(ttext-
   >format,255,0,0));
SDL_BlitSurface(ttext,NULL,screen,&rectangulo);
```

Este código no necesita demasiada explicación. Simplemente comentar que utilizamos la función `sprintf()` para pasar el valor de la puntuación, que es de tipo `int`, a una cadena de texto tipo `char`.

El número de vidas se va a mostrar al jugador de forma gráfica con pequeños avioncitos. Tantos como vidas tengamos. Este código dibuja los avioncitos (almacenados en el sprite `life`) cada 22 píxeles.

```
// Dibuja las vidas del jugador
for (i=1 ; i<=jugador.vidas ; i++) {
   life.setx(10+(i*22));
   life.sety(5);
```

```
   life.draw(screen);
}
```

Es habitual ofrecer al jugador una vida extra cuando consigue algún objetivo concreto. En nuestro caso, vamos a regalar una vida cada 2000 puntos.

```
// ¿vida extra?
if (score % 2000 == 0 && score > 0) {
   score+=10;
   jugador.vidas++;
}
```

Por útimo, lo que le falta a nuestro juego es una portada. Al ejecutar el juego, haremos aparecer una pantalla de presentación y esperaremos la pulsación de una tecla o del botón del joystick para empezar a jugar. Para ello tenemos una variable llamada estado. Si `estado` vale 0, el game loop muestra la pantalla de presentación, si vale 1, quiere decir que estamos jugando y muestra el juego. Este es el código completo del juego.

Figura 8.1. Aspecto de nuestro juego: 1945

```
/*****************************************************************************
 - 1945 -
 (C) 2003 by Alberto Garcia Serrano
 Programación de videojuegos con SDL
 *****************************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <SDL.h>
#include "SDL_mixer.h"
#include "SDL_ttf.h"
#include "csprite.h"


// mapa

#define MAXTILES 10
#define MAXMAP 400

char mapa[401];

SDL_Surface *screen;
SDL_Rect rect;
CFrame fnave;
CFrame fmalo1 1;
CFrame fmalo1_2;
CFrame fmalo1_3;
CFrame fmalo1_4;
CFrame fmalo1_5;
CFrame fmalo2;
CFrame fmalo3;
CFrame bala;
CFrame ex1;
CFrame ex2;
CFrame ex3;
CFrame ex4;
CFrame ex5;
CFrame ex6;
CFrame ex7;
CFrame dis;
CFrame dis2;
CFrame vida;
CFrame pant;
CFrame frsuelo[MAXTILES];
CFrame flevel;
CSprite slevel(1);
CSprite suelo[MAXTILES];
CSprite pantalla(1);
CSprite explode(12);
CSprite nave(1);
CSprite malo2(1);
CSprite malo3(1);
CSprite malo1(5);
CSprite mibala(1);
CSprite dispene(2);
CSprite life(1);
int ciclos,leveltime;
int indice, indice in;
int nexplosiones=7,nmalos=6,nbalas=6,ndispenemigos=7,mibalax[7],mibalay[7];
int done=0,estado=0, conjefe=0, level=1, score=0;
Uint32 ini_milisegundos, fin_milisegundos, frametime;
Sint16 joyx,joyy;
Mix_Music *musica;
Mix Chunk *explosion,*disparo;
SDL Joystick *joystick;
TTF_Font *fuente;
SDL_Surface *ttext;
SDL_Color bgcolor,fgcolor;
SDL_Rect rectangulo;
FILE *f;

struct minave {
   int activo,x,y,vidas,time;
} jugador;
```

```
struct naveenemiga {
   int activo,x,y,dx,dy,tipo,estado,impactos,nframe;
} enemigo[7];

struct explosion {
   int activo,x,y,nframe;
} exp[8];

struct disparo {
   int activo,x,y,dx,dy,estado,time;
} disp[8];



int dx=2,dy=2;

void creaexplosion(int);
void creadispenemigo(int);
void ResetTimeBase(void);
void creajefe(void);
void LoadLevel(int);
int CurrentTime(void);


void DrawScene(SDL_Surface *screen) {
int i,j,t,x,y;
char msg[30];

   // movimiento del scenario (scroll)
   indice_in+=2;
   if (indice_in>=64) {
      indice in=0;
      indice-=10;
   }

   if (indice <= 0) {
      indice=MAXMAP-100; // si llegamos al final, empezamos de nuevo.
      indice in=0;
   }

   //dibujar escenario
   for (i=0 ; i<10 ; i++) {
      for (j=0 ; j<10 ; j++) {
         t=mapa[indice+(i*10+j)];
         // calculo de la posición del tile
         x=j*64;
         y=(i-1)*64+indice_in;

         // dibujamos el tile
         suelo[t].setx(x);
         suelo[t].sety(y);
         suelo[t].draw(screen);
      }
   }


   // Dibuja nave
   if (jugador.activo==1) {
      nave.setx(jugador.x);
      nave.sety(jugador.y);
      nave.draw(screen);
   }

   if (jugador.activo==2 && jugador.time%2==0) {
      nave.setx(jugador.x);
      nave.sety(jugador.y);
      nave.draw(screen);
   }


   // Dibuja enemigo
   for (i=0 ; i<=nmalos ; i++) {
      if (enemigo[i].activo==1) {
         if (enemigo[i].tipo==0) {
            malo1.setx(enemigo[i].x);
            malo1.sety(enemigo[i].y);
            malo1.selframe(enemigo[i].nframe);
```

```
         malo1.draw(screen);
         // ¿ha colisionado con la nave?
         if (malo1.colision(nave) == TRUE && jugador.activo==1) {
            jugador.vidas--;
            creaexplosion(255);
            jugador.time=30;
            jugador.activo=0;
         }

      }

      if (enemigo[i].tipo==1) {
         malo2.setx(enemigo[i].x);
         malo2.sety(enemigo[i].y);
         malo2.draw(screen);
         // ¿ha colisionado con la nave?
         if (malo2.colision(nave) == TRUE && jugador.activo==1) {
            jugador.vidas--;
            creaexplosion(255);
            jugador.time=30;
            jugador.activo=0;
         }

      }

      if (enemigo[i].tipo==3) {
         malo3.setx(enemigo[i].x);
         malo3.sety(enemigo[i].y);
         malo3.draw(screen);
         // ¿ha colisionado con la nave?
         if (malo3.colision(nave) == TRUE && jugador.activo==1) {
            jugador.vidas--;
            creaexplosion(255);
            jugador.time=30;
            jugador.activo=0;
         }
      }

   }
}


// Dibujamos las explosiones
for (i=0 ; i<=nexplosiones; i++) {
   if (exp[i].activo==1) {
      explode.selframe(exp[i].nframe);
      explode.setx(exp[i].x);
      explode.sety(exp[i].y);
      explode.draw(screen);
      exp[i].nframe=exp[i].nframe+1;
      if (exp[i].nframe>=7) {
         exp[i].activo=0;
      }
   }
}

// Dibujamos los disparos
for (i=0 ; i<=nbalas ; i++) {
   if (mibalax[i]) {
      mibala.setx(mibalax[i]);
      mibala.sety(mibalay[i]);
      mibala.draw(screen);

      // ¿hay colisión con alguna nave?
      for (int j=0 ; j<=nmalos ; j++) {
         if (enemigo[j].activo==1) {


            switch (enemigo[j].tipo) {

            case 0:
               // comprobamos impacto con nave tipo 0
               malo1.setx(enemigo[j].x);
               malo1.sety(enemigo[j].y);
               if (mibala.colision(malo1) == TRUE) {
                  // Le hemos dado
                  creaexplosion(j);
```

```
                  enemigo[j].activo=0;
                  mibalax[i]=0;
                  score=score+10;
               }
               break;

            case 1:
               // comprobamos impacto con nave tipo 1
               malo2.setx(enemigo[j].x);
               malo2.sety(enemigo[j].y);
               if (mibala.colision(malo2)) {
                  // Le hemos dado
                  creaexplosion(j);
                  enemigo[j].activo=0;
                  mibalax[i]=0;
                  score=score+10;
               }
               break;

            case 3:
               // comprobamos impacto con nave tipo 3
               malo3.setx(enemigo[j].x);
               malo3.sety(enemigo[j].y);
               if (mibala.colision(malo3)) {
                  // Le hemos dado
                  enemigo[j].impactos++;
                  mibalax[i]=0;
                  // 100 impactos para destruir al jefe
                  if (enemigo[j].impactos >=100 ) {
                     creaexplosion(j);
                     enemigo[j].activo=0;
                     score=score+100;
                     conjefe=2;     // el jefe ha muerto
                     leveltime=100; // tiempo hasta el cambio de nivel
                     level++;
                  }
               }
               break;

            }

         }
      }


   }
}


// Dibujamos los disparos enemigos
for (i=0 ; i<=ndispenemigos ; i++) {
   if (disp[i].activo==1) {
      dispene.setx(disp[i].x);
      dispene.sety(disp[i].y);
      // parpadeo del disparo cada 5 ciclos
      disp[i].time++;
      if (disp[i].time>=5) {
         disp[i].time=0;
         if (disp[i].estado==0) {
            dispene.selframe(1);
            disp[i].estado=1;
         } else {
            dispene.selframe(0);
            disp[i].estado=0;
         }
      }
      dispene.draw(screen);

      // ¿nos han dado?
      if (dispene.colision(nave) && jugador.activo==1) {
         jugador.vidas--;
         creaexplosion(255);
         jugador.time=30;
         jugador.activo=0;
      }
   }
}
```

```
   // Dibuja las vidas del jugador
   for (i=1 ; i<=jugador.vidas ; i++) {
      life.setx(10+(i*22));
      life.sety(5);
      life.draw(screen);
   }

   // dibuja la puntuación
   sprintf(msg,"%d",score);
   ttext = TTF_RenderText_Shaded(fuente,msg,fgcolor,bgcolor);
   rectangulo.y=5;
   rectangulo.x=530;
   rectangulo.w=ttext->w;
   rectangulo.h=ttext->h;
   SDL_SetColorKey(ttext,SDL_SRCCOLORKEY|SDL_RLEACCEL, SDL_MapRGB(ttext->format,255,0,0));
   SDL_BlitSurface(ttext,NULL,screen,&rectangulo);

   // Muestra el nivel en el que estamos jugando
   if ((leveltime>=1) && leveltime--%2) {
      sprintf(msg,"Level %d",level);
      ttext = TTF_RenderText_Shaded(fuente,msg,fgcolor,bgcolor);
      rectangulo.y=240;
      rectangulo.x=310;
      rectangulo.w=ttext->w;
      rectangulo.h=ttext->h;
      SDL SetColorKey(ttext,SDL SRCCOLORKEY|SDL RLEACCEL, SDL MapRGB(ttext-
>format,255,0,0));
      SDL_BlitSurface(ttext,NULL,screen,&rectangulo);
   }


   // Mostramos todo el frame
   SDL_Flip(screen);
}


void muevenaves() {

int i;

   // Esperamos a que el jugador esté preparado para seguir jugando
   // despues de perder una vida.
   if (jugador.activo==2) {
      jugador.time--;
      if (jugador.time<=0) {
         jugador.time=0;
         jugador.activo=1;
      }
   }


   // decrementamos el contador de tiempo de nuestra nave en el caso
   // de haber sido destruidos. Si el tiempo ha pasado, el jugador continua.
   if (jugador.activo==0) {
      jugador.time--;

      if (jugador.time<=0) {
         jugador.time=50;
         if (jugador.vidas==0)
            estado=0;

         jugador.activo=2;
         jugador.x=320;
         jugador.y=400;
      }
   }


   // Actualizamos la posición de las balas
   for (i=0 ; i<=nbalas ; i++) {
      if (mibalax[i]) {
         mibalay[i]=(mibalay[i])-10;

         // ¿ha salido la bala de la pantalla?
         if (mibalay[i]<=10) {
            mibalax[i]=mibalay[i]=0;
```

```
         }

      }
   }



   // Animación (interna) de los sprites
   for (i=0 ; i<=nmalos ; i++) {
      if (enemigo[i].activo == 1) {

         // Animación del enemigo tipo 0 (loop)
         if (enemigo[i].tipo == 0 && enemigo[i].estado == 1) {
            if (enemigo[i].nframe < malo1.frames())
               enemigo[i].nframe++;
         }
      }
   }



   // Movimiento de los enemigos
   for (i=0 ; i<=nmalos ; i++) {
      if (enemigo[i].activo == 1) {
         enemigo[i].x=enemigo[i].x+enemigo[i].dx;
         enemigo[i].y=enemigo[i].y+enemigo[i].dy;

         // Movimiento enemigo tipo 0
         if (enemigo[i].tipo == 0) {

            if (enemigo[i].estado==0 && enemigo[i].y>240) {
               creadispenemigo(i);
               enemigo[i].dy=-3;
               enemigo[i].dx=0;
               enemigo[i].estado=1;
            }

         }

         // Movimiento del jefe
         if (enemigo[i].tipo == 3) {
            // reglas para el movimiento lateral.
            if (enemigo[i].x>500) enemigo[i].dx=-3;
            if (enemigo[i].x<20) enemigo[i].dx=3;

            // un disparo cada 30 ciclos
            if (!(ciclos%30)) {
               creadispenemigo(i);
            }

         }


         // ¿salió la nave de la pantalla?
         if (enemigo[i].y>480 || enemigo[i].y<-50)
            enemigo[i].activo=0;
      }
   }



   // Movimiento de las balas enemigas
   for (i=0 ; i<=ndispenemigos ; i++) {
      if (disp[i].activo==1) {
         disp[i].x=disp[i].x+disp[i].dx;
         disp[i].y=disp[i].y+disp[i].dy;

         // ¿ha salido de la pantalla?
         if (disp[i].y>480)
               disp[i].activo=0;
      }
   }

}


void creadisparo() {
```

```
   int libre=-1;

   // ¿Hay alguna bala libre?
   for (int i=0 ; i<=nbalas ; i++) {
      if (mibalax[i]==0)
         libre=i;
   }

   // Hay una bala
   if (libre>=0) {
      mibalax[libre]=nave.getx();
      mibalay[libre]=nave.gety()-15;
   }

   // sonido del disparo
   Mix_PlayChannel(-1,disparo,0);
}


void creaenemigo() {
   int libre=-1;

   // ¿Hay algún enemigo libre?
   for (int i=0 ; i<=nmalos ; i++) {
      if (enemigo[i].activo==0)
         libre=i;
   }

   // Hay un enemigo dispuesto
   if (libre>=0) {
      enemigo[libre].activo=1;
      enemigo[libre].nframe=0;
      enemigo[libre].x=rand();   // Pos.X aleatoria
      if (enemigo[libre].x > 640)
         enemigo[libre].x=(int)enemigo[libre].x % 640;
      enemigo[libre].tipo=rand();   // tipo de enemigo aleatorio
      if (enemigo[libre].tipo >= 2)
         enemigo[libre].tipo=(int)enemigo[libre].tipo % 2; // 2 tipos de enemigos (0,1)

      // tipo caza
      if (enemigo[libre].tipo==0) {
         enemigo[libre].y=-30;
         enemigo[libre].dx=0;
         enemigo[libre].dy=5;
         enemigo[libre].estado=0;
      }

      // tipo kamikaze
      if (enemigo[libre].tipo==1) {
         enemigo[libre].y=-30;
         if (enemigo[libre].x>nave.getx()) {
            enemigo[libre].dx=-3;
         } else {
            enemigo[libre].dx=3;
         }
         enemigo[libre].dy=5;
         enemigo[libre].estado=0;
      }

   }
}



void creajefe() {
   int libre=1;

   enemigo[libre].activo=1;
   enemigo[libre].tipo=3;
   enemigo[libre].y=100;
   if (enemigo[libre].x>nave.getx()) {
      enemigo[libre].dx=-3;
   } else {
      enemigo[libre].dx=3;
   }
   enemigo[libre].dy=0;
```

```
   enemigo[libre].estado=0;
   enemigo[libre].impactos=0;

}


void creaexplosion(int j) {

   int libre=-1;

   for(int i=0; i<=nexplosiones; i++) {
      if (exp[i].activo==0)
         libre=i;
   }

   if (libre>=0) {
      if (j!=255) { // si j = 255 la explosión es para el jugador
         exp[libre].activo=1;
         exp[libre].nframe=1;
         exp[libre].x=enemigo[j].x;
         exp[libre].y=enemigo[j].y;
      } else {
         exp[libre].activo=1;
         exp[libre].nframe=1;
         exp[libre].x=jugador.x;
         exp[libre].y=jugador.y;
      }
   }

   // sonido de explosion
   Mix_PlayChannel(-1,explosion,0);
}


void creadispenemigo(int j) {
   int libre=-1;

   for(int i=0; i<=ndispenemigos; i++) {
      if (disp[i].activo==0)
         libre=i;
   }

   if (libre>=0) {
      disp[libre].activo=1;
      if (enemigo[j].x>nave.getx()) {
         disp[libre].dx=-3;
      } else {
         disp[libre].dx=3;
      }
      disp[libre].dy=3;
      disp[libre].x=enemigo[j].x+30;
      disp[libre].y=enemigo[j].y+20;
      disp[libre].estado=0;
      disp[libre].time=0;
   }
}


void InitEstados() {

int i;

   jugador.activo=1;
   jugador.vidas=3;
   jugador.x=320;
   jugador.y=400;

   // Inicializamos el array de balas
   for (i=0 ; i<=nbalas ; i++) {
      mibalax[i]=0;
      mibalay[i]=0;
   }

   // Inicializamos el array de enemigos
   for (i=0 ; i<=nmalos ; i++) {
      enemigo[i].activo=0;
   }
```

```
   // Inicializamos el array de explosiones
   for (i=0; i<=nexplosiones ; i++) {
      exp[i].activo=0;
   }

   // Inicializamos el array de disparos enemigos
   for(i=0; i<=ndispenemigos; i++) {
      disp[i].activo=0;
   }

   // inicializa colores para el texto
   fgcolor.r=200;
   fgcolor.g=200;
   fgcolor.b=10;

   bgcolor.r=255;
   bgcolor.g=0;
   bgcolor.b=0;

   // Posición inicial en el mapa
   indice=MAXMAP-100;
   indice_in=0;

   ciclos=0;
   conjefe=0;
}


void LoadLevel(int l) {

   InitEstados();

   leveltime=100;

   // Cargar mapa (10x40)
   char tmplevel[50];
   int i, c, ntiles;

   sprintf(tmplevel,"levels/level%d/map.map",l);
   if((f=fopen(tmplevel,"r")) != NULL) {
      c=fread(mapa,MAXMAP,1,f);
      fclose(f);
   }

   // Cargar tiles del suelo
   sprintf(tmplevel,"levels/level%d/level.dsc",l);
   if((f=fopen(tmplevel,"r")) != NULL) {
      fgets(tmplevel,255,f);
      ntiles=atoi(tmplevel);
      fclose(f);
   }

   for (i=0 ; i<=ntiles ; i++) {
      sprintf(tmplevel,"levels/level%d/tile%d.bmp",l,i);
      frsuelo[i].load(tmplevel);
      suelo[i].addframe(frsuelo[i]);
   }
}


void inicializa() {
   // carga la funte de letra
   fuente = TTF_OpenFont("ariblk.ttf",20);

   // Activa el Joystick
   if (SDL_NumJoysticks() >= 1) {
      joystick = SDL_JoystickOpen(0);
      SDL_JoystickEventState(SDL_ENABLE);
   }

   // carga la musica y los sonidos
   musica = Mix_LoadMUS("sounds/rwvalkyr.mid");
   explosion = Mix_LoadWAV("sounds/explosion.wav");
   disparo = Mix_LoadWAV("sounds/disparo.wav");
}
```

```
void finaliza() {

   // destruimos la fuente de letra
   TTF_CloseFont(fuente);

   // destrimos la musica
   Mix_FreeMusic(musica);

   // cerramos el joystick
   if (SDL_JoystickOpened(0)) {
      SDL_JoystickClose(joystick);
   }

}


int InitSprites() {

int i;

   // Inicializamos el generador de números aleatorios
   srand( (unsigned)time( NULL ) );

   // Inicializamos el array de balas
   for (i=0 ; i<=nbalas ; i++) {
      mibalax[i]=0;
      mibalay[i]=0;
   }


   // Inicializamos el array de enemigos
   for (i=0 ; i<=nmalos ; i++) {
      enemigo[i].activo=0;
   }

   // Inicializamos el array de explosiones
   for (i=0; i<=nexplosiones ; i++) {
      exp[i].activo=0;
   }


   pant.load("pics/pantalla.bmp");

   pantalla.addframe(pant);

   ex1.load("pics/explode1.bmp");
   explode.addframe(ex1);

   ex2.load("pics/explode2.bmp");
   explode.addframe(ex2);

   ex3.load("pics/explode3.bmp");
   explode.addframe(ex3);

   ex4.load("pics/explode4.bmp");
   explode.addframe(ex4);

   ex5.load("pics/explode5.bmp");
   explode.addframe(ex5);

   ex6.load("pics/explode6.bmp");
   explode.addframe(ex6);

   ex7.load("pics/explode7.bmp");
   explode.addframe(ex7);

   dis.load("pics/dispene.bmp");
   dispene.addframe(dis);

   dis2.load("pics/dispene2.bmp");
   dispene.addframe(dis2);

   fnave.load("pics/minave.bmp");
   nave.addframe(fnave);
```

```
   fmalo1_1.load("pics/enemy1_1.bmp");
   malo1.addframe(fmalo1 1);

   fmalo1_2.load("pics/enemy1_2.bmp");
   malo1.addframe(fmalo1_2);

   fmalo1_3.load("pics/enemy1_3.bmp");
   malo1.addframe(fmalo1 3);

   fmalo1_4.load("pics/enemy1_4.bmp");
   malo1.addframe(fmalo1_4);

   fmalo1 5.load("pics/enemy1 5.bmp");
   malo1.addframe(fmalo1 5);

   fmalo2.load("pics/nave2.bmp");
   malo2.addframe(fmalo2);

   fmalo3.load("pics/gordo.bmp");
   malo3.addframe(fmalo3);

   bala.load("pics/balas.bmp");
   mibala.addframe(bala);

   vida.load("pics/life.bmp");
   life.addframe(vida);


   return 0;

}


void ResetTimeBase() {

   ini milisegundos=SDL GetTicks();
}


int CurrentTime() {

   fin milisegundos=SDL GetTicks();
   return fin milisegundos-ini milisegundos;
}


int main(int argc, char *argv[]) {

SDL Event event;
Uint8 *keys;
int salir=0;


   if (SDL Init(SDL INIT VIDEO|SDL INIT AUDIO|SDL INIT JOYSTICK) < 0) {
      printf("No se pudo iniciar SDL: %s\n",SDL_GetError());
      return 1;
   }

   screen = SDL SetVideoMode(640,480,24,SDL HWSURFACE|SDL DOUBLEBUF|SDL FULLSCREEN);
   if (screen == NULL) {
      printf("No se puede inicializar el modo gráfico: \n",SDL_GetError());
      return 1;
   }

   atexit(SDL Quit);

   if (Mix_OpenAudio(MIX_DEFAULT_FREQUENCY,MIX_DEFAULT_FORMAT,1,4096) < 0) {
      printf("No se pudo iniciar SDL_Mixer: %s\n",Mix_GetError());
      return 1;
   }

   atexit(Mix CloseAudio);

   if (TTF_Init() < 0) {
      printf("No se pudo iniciar SDL_ttf: %s\n",SDL_GetError());
      return 1;
```

```
}

atexit(TTF_Quit);

inicializa();

InitSprites();


while (done == 0) {

   switch (estado) {

   case 0:

      InitEstados();
      pantalla.setx(0);
      pantalla.sety(0);
      pantalla.draw(screen);

      SDL_Flip(screen);

      // desactivamos la música (si estaba activada)
      Mix_HaltMusic();

      while (estado==0 && done==0) {

         while (SDL_PollEvent(&event)) {

            if (event.type == SDL_QUIT) {done=1;}

            if (event.type == SDL_KEYDOWN || event.type == SDL_JOYBUTTONDOWN) {

               if (event.key.keysym.sym == SDLK_ESCAPE) {
                  done=1;
               } else {
                  score=0;
                  estado=1;
                  LoadLevel(1);

                  // Activamos la música
                  Mix_PlayMusic(musica,-1);

               }

            }

         }
      }

      break;

   case 1:

      // Inicializamos el timer para control de tiempo.
      ResetTimeBase();

      ciclos++;

      // cada 20 ciclos se crea un enemigo
      if ((!(ciclos % 20)) && conjefe==0) {
         creaenemigo();
      }

      // ¿final de fase?
      if (ciclos>=5000 && conjefe==0) {
         // creamos la nave jefe.
         creajefe();
         conjefe=1;
      }


      // tiempo desde que se destruye al jefe
      // hasta que cambiamos de nivel.
      if ((conjefe == 2) && (leveltime-- <= 0)) {
         if (level>2) level=1;
         LoadLevel(level);
```

```
         }

         muevenaves();
         DrawScene(screen);

         // ¿vida extra?
         if (score % 2000 == 0 && score > 0) {
            score+=10;
            jugador.vidas++;
         }


         // consultamos el estado del teclado
         keys=SDL GetKeyState(NULL);

         // consultamos el estado del joystick
         SDL_JoystickUpdate();

         joyx = SDL_JoystickGetAxis(joystick, 0);
         joyy = SDL JoystickGetAxis(joystick, 1);

         if ((keys[SDLK_UP] || joyy < -10) && (jugador.y > 0)) {jugador.y=jugador.y-(5);}
         if ((keys[SDLK_DOWN] || joyy > 10) && (jugador.y < 460))
{jugador.y=jugador.y+(5);}
         if ((keys[SDLK_LEFT] || joyx < -10) && (jugador.x > 0)) {jugador.x=jugador.x-(5);}
         if ((keys[SDLK RIGHT] || joyx > 10) && (jugador.x < 620))
{jugador.x=jugador.x+(5);}
         if (keys[SDLK_LSHIFT] && jugador.activo!=0) {creadisparo();}


         while (SDL_PollEvent(&event)) {

            // evento del boton del joystick
            if (event.type ==SDL_JOYBUTTONDOWN && jugador.activo != 0) {
               creadisparo();
            }

            // evento de cierre del programa
            if (event.type == SDL_QUIT) {done=0;}

            // evento pulsación de tecla
            if (event.type == SDL_KEYDOWN) {

               if (event.key.keysym.sym == SDLK ESCAPE) {estado=0;}

            }

         }

         // Esperamos a que transcurran al menos 30 ms para
         // generar el siguiente frame, y evitar así
         // que el juego se acelere en máquinas muy rápidas.

         do {
            frametime=CurrentTime();
         } while (frametime<30);

         break;
      }
   }

   finaliza();

   return 0;
}
```

¿Y ahora que?

Ahora te queda la mayoría del camino por recorrer. Si este libro cumple bien el objetivo con el que fue escrito, espero haberte hecho dar los primeros pasos en la dirección correcta. Me voy a permitir un último consejo: haz tu propio juego. No es necesario que sea muy complejo, lo realmente importante es que lo termines. Y recuerda que la mejor forma de aprender a programar es ¡programando! Tampoco te hará mal leer código ajeno. Explora todo el código que puedas. Hay montones de juegos de código abierto.

Vivimos unos buenos tiempos para los que tenemos curiosidad y ganas de aprender. Internet nos brinda acceso a gran cantidad de información que está disponible a un click de distancia. El apéndice C puedes encontrar numerosos recursos sobre programación de videojuegos. También hay muy buenos libros en el mercado, quien sabe, puede que nos encontremos de nuevo en una segunda parte de este libro. Según te vayas introduciendo en este mundo descubrirás que cada vez te falta más y más por aprender. No te desanimes. La programación de juegos está compuesta por muy diferentes aspectos, como la IA, gráficos 3D, programación de audio, etc... y no puedes ser un experto en todos.

Quizás el siguiente paso sea entrar en el mundo de las tres dimensiones y SDL ofrece un buen soporte para OpenGL. Todo lo aprendido en este libro es la base y es aplicable en los juegos 3D, aunque, por supuesto, tendrás que seguir aprendiendo y recorriendo el camino antes de crear tu primer juego en tres dimensiones. No utilces atajos, es peor, créeme.

Cuando lleves algún tiempo en esto, es probable que quieras dedicarte profesionalmente a la creación de los videojuegos. Aquí los curriculums valen de poco, al menos, si estás intentando meter la cabeza en el mundillo. Tu carta de presentación será tu trabajo, así que prepara algunas demos de tus juegos y prepárate a mandarlas a todas las compañías que puedas encontrar. Te advierto que, o lo haces por vocación, o lo pasarás mal. Crear un juego comercial es como una carrera de fondo, tienes que disfrutar realmente con lo que haces. Piensa que un juego comercial medio puede llevar un año de trabajo para su finalización, pero si es lo que quieres... hazlo y no te dejes influir por lo que nadie te cuente.

Dice Oscar Wilde en "*El retrato de Dorian Gray*":

"Porque influir en una persona es darle la propia alma. Esa persona deja de pensar sus propias ideas y de arder con sus pasiones. Sus virtudes dejan de ser reales. Sus pecados, si es que los pecados existen, son prestados. Se convierte en el eco de la música de otro, en un actor que interpreta un papel que no se ha escrito para él. La finalidad de la vida es el propio desarrollo. Alcanzar la plenitud de la manera más perfecta posible, para eso estamos aquí."

Arde con tus propias pasiones y...

¡Disfruta!

Apéndice

A

Instalación de SDL

La dirección web oficial del proyecto SDL es *http://www.libsdl.org*. Allí podrás encontrar gran cantidad de información, documentación y aplicaciones desarrolladas con SDL. Las direcciones directas para descargar SDL y sus librerías auxiliares son:

Librería SDL	http://www.libsdl.org/download-1.2.php
SDL_mixer	http://www.libsdl.org/projects/SDL_mixer/
SDL_ttf	http://www.libsdl.org/projects/SDL_ttf/
SDL_image	http://www.libsdl.org/projects/SDL_image/

En este apéndice vamos a ver como utilizar estas librerías tanto en Windows (con VC++) como en Linux (con GCC).

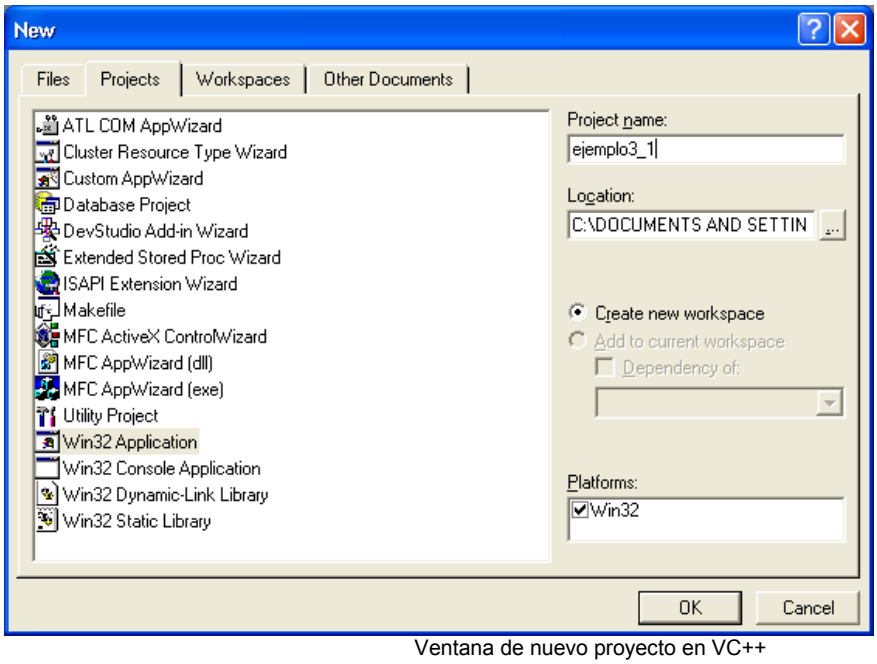
Windows (VC++)

Lo primero que necesitamos es tener instalado el compilador de C. Supondré que utilizas Visual C++ 6.0, ya que es el más extendido y utilizado en entornos Windows. Si tienes otro compilador, los pasos a realizar serán similares, aunque ya dependerá de cada caso. Lo mejor es consultar la documentación de tu compilador.

El primer paso es descargar las librerías necesarias. Asegúrate que sean las *development libraries*. Si utilizas VC++ existe una versión especial de las librerías, por ejemplo, el archivo que debes descargar para VC es *SDL-devel-1.2.6-VC6.zip*. Una vez descargado descomprímelo en el directorio raiz. Ahora debes tener un directorio llamado *\SDL-1.2.6*. En lugar de 1.2.6, podrían ser otros valores dependiendo de la versión de la librería.

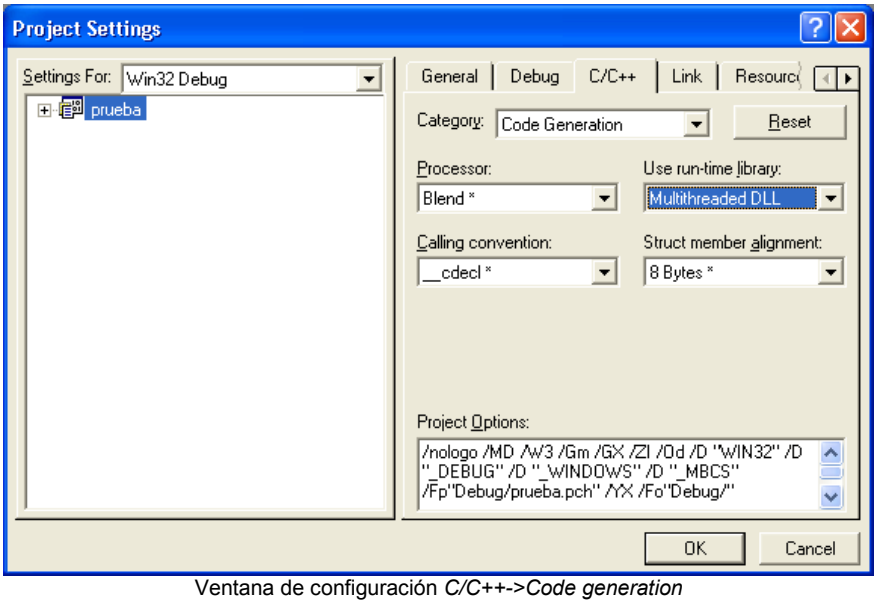
Arranca VC++ y selecciona la opción *new...* del menú *file*. Podrás ver la ventana de nuevo proyecto. En la pestaña *Project* selecciona *WIN32 Application*, escribe el nombre del proyecto en el cuadro de texto *project name*, y selecciona el directorio donde quieres almacenarlo en *location*. Seguidamente pulsa el botón OK.

En la siguiente ventana nos pregunta que tipo de aplicación queremos crear. Seleccionamos un proyecto vacío (An empty project) y pulsamos el botón finish. Ya tenemos nuestro proyecto creado. Ahora vamos a configurar el proyecto para poder utilizar las librerías de SDL.



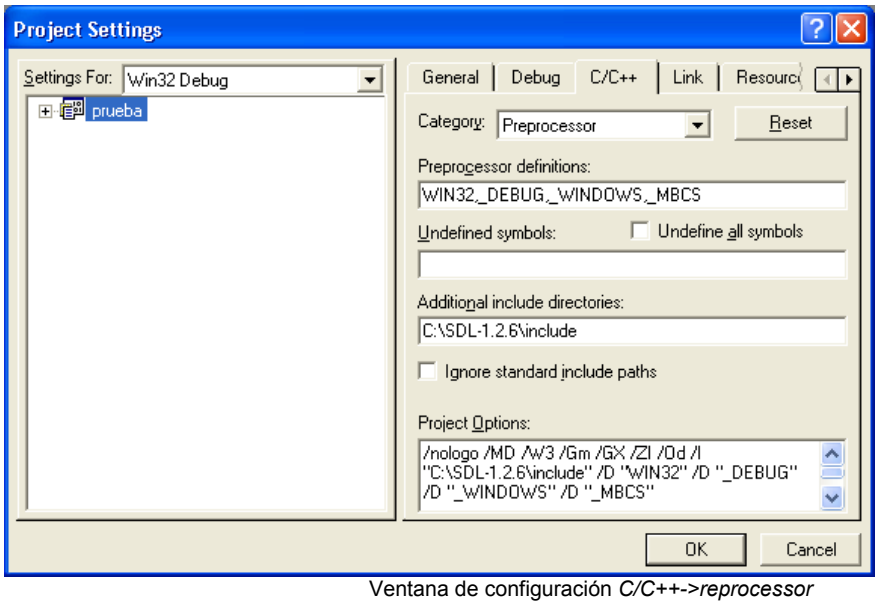
Ventana de nuevo proyecto en VC++

Selecciona la opción *Settings...* del menú *Project*. Verás como aparece una ventana de configuración. Selecciona la pestaña C/C++, aparecerá un campo desplegable llamado *Category*. Selecciona *code generation* en este desplegable. En el desplegable llamado *Use run-time library*, selecciona *Multithreaded DLL*.



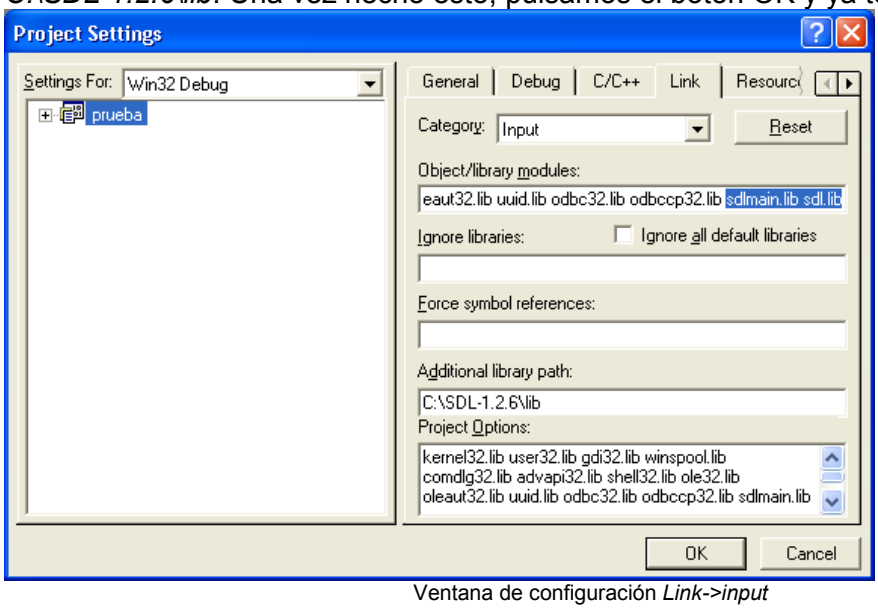
Ventana de configuración *C/C++->Code generation*

El siguiente paso es seleccionar la opción *Preprocessor* en el desplegable *Category*. En el campo *Additional include directories*, incluimos la ubicación del directorio include que hay dentro del directorio de SDL. En este caso es *C:\SDL-1.2.6\include*.



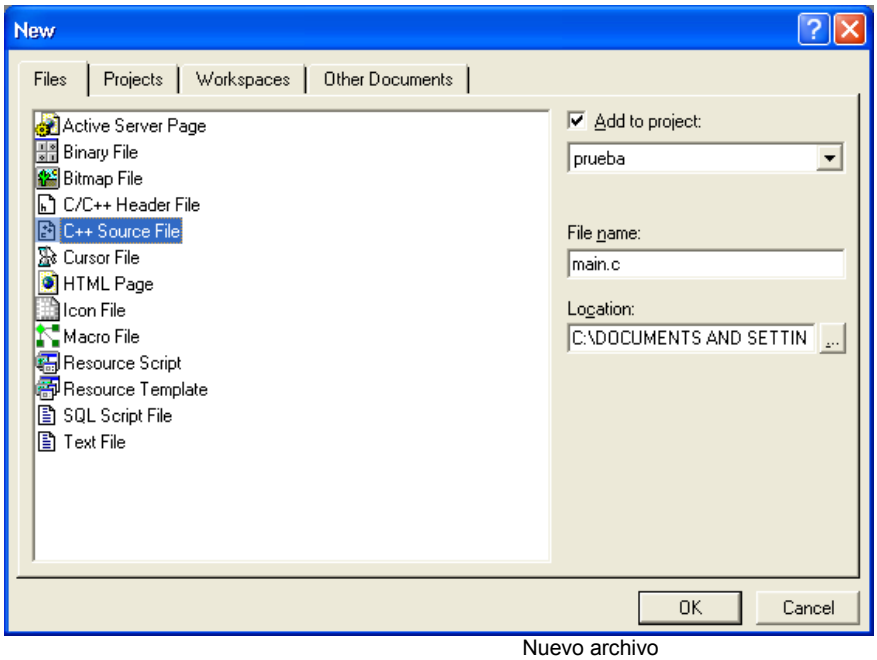
Ventana de configuración *C/C++->reprocessor*

Ahora vamos a configurar el enlazador (linker) del compilador para que pueda enlazar las librerías. Pulsamos la pestaña Link, y seleccionamos *input*. El el cuadro de texto *Object/library modules* añadimos *sdlmain.lib* y *sdl.lib*. En caso de que queramos utilizar otras librerías, como por ejemplo, SDL_mixer, hemos de añadirlas también aquí. En el campo *Aditional library path*, vamos a añadir la ubicación de las librerías. En este caso *C:\SDL-1.2.6\lib*. Una vez hecho esto, pulsamos el botón OK y ya tenemos todo listo.



Ventana de configuración *Link->input*

Vamos a añadir código a nuestro proyecto. Selecciona la opción *New...* del menú *file*. Veremos una ventana que nos permite seleccionar el tipo de archivo. Asegúrate que está seleccionada la pestaña *File*. En la ventana selecciona *C++ Source File*. En el campo *File name*, introduce el nombre del archivo. En este caso lo llamaremos *main.c*. Seguidamente pulsamos OK.



Nuevo archivo

Ahora tendremos una ventana de edición lista para empezar a teclear. Introduce el siguiente código que utilizaremos para probar la librería.

```
#include <stdio.h>
#include <stdlib.h>
#include <SDL.h>


int main(int argc, char *argv[]) {

 SDL_Surface *screen;
 SDL_Event event;
 int done = 0;

  atexit(SDL_Quit);

  // Iniciar SDL
  if (SDL_Init(SDL_INIT_VIDEO) < 0) {
    printf("No se pudo iniciar SDL: %s\n",SDL_GetError());
    exit(1);
  }

  // Activamos modo de video
  screen = SDL_SetVideoMode(640,480,24,SDL_HWSURFACE);
  if (screen == NULL) {
    printf("No se puede inicializar el modo gráfico:
\n",SDL_GetError());
    exit(1);
  }


  // Esperamos la pulsación de una tecla para salir
  while(done == 0) {
    while ( SDL_PollEvent(&event) ) {
      if ( event.type == SDL_KEYDOWN )
        done = 1;
    }
```

```
    }

  return 0;
}
```

Una vez introducido el código, pulsamos F7 para compilar el programa. Si todo va bien, VC++ nos informará que no ha habido ningún error. Si no es así repasa el código.

Como por defecto VC++ está en modo debug, el compilador nos habrá creado un directorio llamado *Debug* en el directorio donde se encuentra el proyecto. Dentro de este directorio encontramos un ejecutable. Éste es nuestro programa ya compilado, sólo nos resta un paso. Para que los programas que utilizan SDL funcionen, hemos de copiar el archivo SDL.dll en el mismo directorio donde se encuentre el programa ejecutable (o añadirlo al directorio \windows\system32). Ya está listo para su ejecución pulsando doble click sobre él. Si todo ha ido bién, verás una ventana en negro. Pulsa cualquier tecla para cerrar el programa.

Observa que si actualizas la versión de SDL, o intentas cargar un proyecto que utilice otra versión distinta a la que tienes instalada, tendrás que actualizar la ubicación de las cabeceras y las librerías en el proyecto. Por ejemplo, si antes tenías tus cabeceras en C:\SDL-1.2.4\include y te actualizas a la versión 1.2.6 de SDL, tendrás que cambiar la ubicación en la configuración a C:\SDL-1.2.6\include.

La instalación de las librerías auxiliares es muy sencilla. Descarga la versión especifica para VC++, y copia los archivos de cabecera (*.h) al directorio *include* de SDL (C:\SDL-1.2.6\include) y las librerías (*.lib) al directorio *lib* de SDL (C:\SDL-1.2.6\lib). Asegúrate de copiar el archivo DLL de la librería (por ejemplo SDL_mixer.dll) en el directorio donde se encuentre el archivo ejecutable.

Linux

Tal y como hicimos con Windows, lo primero que tenemos que hacer es localizar el paquete que nos conviene descargar. Si tu distribución Linux está basada en Red Hat o trabaja con paquetes en formato RPM, tendrás que bajarte los paquetes que finalizan con la extensión .rpm. La instalación de un paquete RPM se hace de la siguiente manera (siempre como usuario root).

rpm –ivh nombre_paquete

Por ejemplo, si el paquete que has descargado es *SDL-1.2.6-1.i386.rpm*, la línea que realiza la instalación es:

```
rpm –ivh SDL-1.2.6-1.i386.rpm
```

Si tu distribución está basada en Debian, podrás utilizar la utilidad apt para descargar e instalar SDL. La siguiente línea descarga e instala SDL.

```
apt-get install libsdl1.2-dev
```

Esta línea podría variar dependiendo de la última versión de SDL disponible. Es conveniente ejecutar la línea siguiente antes para asegurarte que descargas la última versión.

```
apt-get update
```

Seguidamente, para conocer que versiones y que librerías para SDL hay disponibles utliza la siguiente línea.

```
apt-cache search sdl
```

Una vez instalado SDL, las librerías deberían estar instaladas en el directorio /usr/lib, y los archivos de cabecera en /usr/include/SDL.

No vamos a entrar en detalle en el funcionamiento del compilador GCC, simplemente te mostraré las líneas necesarias para realizar la compilación.

Para realizar una compilación utilizando librerías dinámicas, compila el programa de la siguiente forma.

```
gcc -L/usr/lib -lSDL -lpthread -I/usr/include/SDL -D_REENTRANT main.cpp csprite.cpp -o
ejemplo6_1
```

El modificador `-L` permite indicar la ubicación de las librerías, y el modificador `-I` indica la ubicación de los archivos de cabecera. El modificador `-l`, nos permite indicar al compilador qué librerías queremos enlazar con nuestro programa. Si quisieramos enlazar la librería principal de SDL y también SDL_mixer utilizaríamos `-lSDL -lSDL_mixer`. Por último, indica los archivos fuente que quieres compilar (`main.cpp` y `csprite.cpp`) y el nombre del archivo ejecutable que quieres generar después del modificador `-o`. Si no entiendes el resto de modificadores no te preocupes, déjalos tal cual, ya que si no, no funcionará el archivo ejecutable.

Para realizar una compilación enlazando las librerías estáticas, hazlo de la siguiente forma.

```
gcc -L/usr/lib -lSDL -lpthread -lm -L/usr/X11R6/lib -lX11 -lXext -ldl
-I/usr/include/SDL -D_REENTRANT main.cpp csprite.cpp
-o ejemplo6_1
```

Para ejecutar el programa ya compilado usamos

```
./ejemplo6_1
```

Apéndice

B

De C a C++

Que tengas este libro entre tus manos, significa que tienes unos conocimientos mínimamente razonables de programación. Probablemente estarás familiarizado con el lenguaje C. En el presente capítulo voy a tratar de hacer una introducción a la programación orientada a objetos en C++. Evidentemente, para hacer una buena introducción necesitaríamos todo un libro. En la bibliografía hay algunos muy buenos que te servirán para profundizar. El objetivo, pues, de este apéndice es acercar la POO en C++ a los programadores con conocimientos previos de C. Nos centraremos en los aspectos más relevantes y, desgraciadamente, por la extensión limitada de la presente obra, dejaremos de lado las características más complejas o menos útiles de este lenguaje. Ni que decir tiene que si ya eres un experto en C++, puede prescindir alegremente este apéndice.

Puede que en estos momentos te estés preguntando si es realmente necesario utilizar C++ y si C no es suficiente. Rotundamente sí. Puedes acometer el desarrollo de juegos razonablemente complejos en C, sin embargo, la POO nos aporta ciertamente una serie de ventajas que hemos de tener muy en cuenta a la hora de sentarnos a programar, como:

- Una mayor abstracción de datos.
- Más facilidad para la reutilización de código.
- Mantenimiento y extensión de las aplicaciones más fácil.
- Mayor potencia (herencia y polimorfismo).
- Modelado de problemas reales de forma más directa.

Quizás todo esto te suene un poco ajeno a la forma a la que estás acostumbrado a programar, e incluso puede que ni te suenen algunos de estos conceptos, pero al finalizar el capítulo, deberán serte familiares.

Todo esto suena bien, pero ¿qué es un objeto? Trataré, sin entrar en demasiados formalismos, explicarlo de la forma más intuitiva posible. Si te pido que pienses en un objeto, seguramente pensarás en un lápiz, una mesa, unas gafas de sol, un coche o cualquier otra cosa que caiga dentro de tu radio de visión. Esta es la idea intuitiva de objeto. Algo físico y material. En POO, el concepto de objeto no es muy diferente. Una de las diferencias básicas evidentes es que un objeto en C++ puede hacer referencia a algo abstracto.

Como en el ejemplo del coche, un objeto puede estar compuesto por otra *clase* de objetos, como ruedas, carrocería, etc... Este concepto de *clase* de objeto es importante. Un objeto siempre pertenece a una clase de objetos. Por ejemplo, todas las ruedas, con independencia de su tamaño, pertenecen a la clase “rueda”. Hay muchos objetos rueda diferente que pertenecen a la clase rueda. Cada una de estas ruedas diferentes se llaman *instancias* de la clase “rueda”. Tenemos, pues, instancias de la clase rueda que son ruedas de camión, ruedas de coches o ruedas de moto.

Volvamos al ejemplo del coche. Vamos a definir otra clase de objeto, la clase "Coche". La clase coche define a "algo" que está compuesto por objetos (instancias) de la clase rueda, la clase carrocería, la clase volante, etc... Ahora vamos a crear un objeto de la clase coche, al que llamaremos coche_rojo. En este caso hemos *instanciado* un objeto de la clase coche y hemos definido uno de sus *atributos*, el color, al que hemos dado el valor de rojo. Vemos pues que un objeto puede poseer atributos. Sobre el objeto coche podemos definir también acciones u operaciones posibles. Por ejemplo, el objeto coche, entre otras cosas, puede realizar las operaciones de acelerar, frenar, girar a la izquierda, etc... Estas operaciones que pueden ser ejecutadas sobre un objeto se llaman *métodos* del objeto.

Podemos hacer ya una primera definición de lo que es un objeto. Es la *instancia* de una clase de objeto concreta, que está compuesta por *atributos* y *métodos*. Esta definición nos muestra ya una de las tres principales características que definen a la POO. Me refiero al *Encapsulamiento*, que no es, ni más ni menos, que la capacidad que tiene un objeto de contener datos (atributos) y código (métodos).

Clases y objetos

Veamos en la práctica como se declara una clase y se instancia un objeto en C++. Podríamos afirmar que, en C++, una clase es algo muy similar a una estructura de C en la que además de almacenar datos, podemos añadir las funciones (métodos) que harán uso de estos datos. Siguiendo con el ejemplo anterior, vamos a declarar la clase "coche".

```
// Declaración de la clase coche
class Coche {
   // Atributos de la clase coche
   int velocidad;

   // Métodos de la clase coche
   void acelerar(int velocidad);
   void frenar();
};
```

En este ejemplo hemos declarado la clase coche. Lo primero que puede chocar al programador de C es la forma de incluir comentarios en el código. C++ permite seguir usando comentarios al estilo de C, pero también añade una nueva forma de hacerlo mediante las dos barras //. C++ interpreta que lo que sigue a las dos barras y hasta el final de la línea es un comentario.

La declaración de la clase coche muestra claramente dos partes diferenciadas. Por un lado, tenemos la declaración de los atributos de la clase (también llamados datos miembro de la clase), que en este caso describe la velocidad del coche. Podemos decir que las variables miembro de una clase definen el estado del objeto (en este caso el estado del coche está formado exclusivamente por su velocidad). Ni que decir tiene que esto es un ejemplo muy simplificado y que una clase que tratara de describir el comportamiento de un coche sería mucho más compleja. La segunda parte de la declaración contiene los métodos (funciones) que nos van a permitir realizar operaciones o acciones sobre el objeto. En el caso de la clase coche hemos definido un método `acelerar()` que nos va a permitir acelerar el coche hasta una determinada velocidad. El método `frenar()` va a hacer que la velocidad sea 0, es decir, que el coche se pare. Esta es la forma en que se implementa un método en C++. Las siguientes líneas implementan el método `frenar()` y `acelerar()`.

```
void Coche::frenar() {
   // Ponemos a 0 el valor del atributo velocidad
   velocidad = 0;
}
```

```
void Coche::acelerar(int velocidad) {
   // Actualizamos la velocidad
   this.velocidad = velocidad;
}
```

Se puede observar que declarar un método es muy similar a declarar una función en C, salvo que es necesario indicar a qué clase corresponde el método. Esto se hace añadiendo el nombre de la clase seguido de dobles dos puntos antes del nombre del método. En el método acelerar, puedes observar que en vez de nombrar directamente a la variable `velocidad`, hemos utilizado `this.velocidad`. El motivo, como seguramente estarás suponiendo, es que la variable que hemos pasado como parámetro al método se llama igual que la variable miembro de la clase. Con `this.velocidad` informamos a C++ de que nos referimos a la variable miembro, en caso contrario interpretará que nos referimos a la variable recibida como parámetro.

Ya sabemos cómo declarar una clase y como implementarla. Nos resta conocer cómo utilizarla. No podemos utilizar una clase directamente. Una clase es como un patrón a partir del cual se crean (instancian) objetos deribados de ella. La forma de crear un objeto coche es declarando el objeto, tal y como haríamos con una variable.

```
Coche micoche;
```

Ahora sí tenemos un objeto llamado `micoche`, que es una instancia de la clase `Coche` y que podemos empezar a utilizar.

Para hacer uso de un método del objeto micoche, procedemos de la siguiente forma.

```
// Primero aceleramos hasta 100 KM/H
micoche.acelerar(100);

// Y despues frenamos
micoche.frenar()
```

Como ves, el acceso a un método es simple, sólo hay que anteponer el nombre del objeto al que nos referimos seguido de un punto y el nombre del método. De la misma forma podemos acceder directamente a los atributos o variables miembro del objeto.

```
// Establecemos la velocidad directamente
micoche.velocidad = 100;
```

Tengo que decir que ésta no es la manera más conveniente de proceder. La mejor decisión de diseño de clases es esconder las variables miembro para que no sean visibles desde fuera de la clase, y ofrecer los métodos necesarios para manejarlas. Así, siempre podemos filtrar desde los métodos los valores que toman los atributos. No queremos que nadie haga algo como `micoche.velocidad = 1000`. No sería nada bueno para el coche. Para especificar quién tiene acceso y quién no a las variables miembro y a los métodos utilizamos los modificadores de acceso. Hay tres modificadores:

- **Public**. Permite el acceso sin restricciones.
- **Private**. Sólo se permite el acceso desde dentro de la clase.
- **Protected**. Permite sólo el acceso desde dentro de la clase o desde una subclase.

Un poco más abajo veremos lo que es una subclase, pero antes veamos un ejemplo de uso de los modificadores.

```
// Declaración de la clase coche
class Coche {
```

```
   // Atributos de la clase coche
   private:
   int velocidad;

   // Métodos de la clase coche
   public:
   void acelerar(int velocidad);
   void frenar();
};
```

Al declarar la clase de esta forma, el atributo `velocidad` no es accesible desde fuera de la clase, es decir, no podríamos hacer algo como `micoche.velocidad=100`. La unica forma de escribir en la variable coche es hacerlo mediante los métodos `acelerar()` y `frenar()`.

Al instanciar un objeto, nunca podemos estar seguro de cual es su estado (el valor de sus atributos). Necesitamos un mecanismo que nos permita inicializar un objeto con los valores deseados. Esto podemos hacerlo mediante el uso de constructores. Un constructor es un método especial de la clase que se ejecuta de forma automática al instanciar la clase. Veámoslo sobre un ejemplo.

```
// Declaración de la clase coche
class Coche {
   // Atributos de la clase coche
   private:
   int velocidad;

   // Métodos de la clase coche
   public:
   Coche();
   void acelerar(int velocidad);
   void frenar();
};

Coche::Coche() {
   // Inicializamos la velocidad a 0
   velocidad = 0;
}
```

Observa que el contructor tiene exactamente el mismo nombre que la clase (incluida la primera letra mayúscula). Otra cosa a tener en cuenta es que un constructor no devuelve nigún valor, ni siquiera void. Cuando instanciemos un objeto de la clase Coche, se ejecutará el código del constructor de forma automática, sin necesidad de invocar el método constructor. En este caso, inicializa la variable velocidad a 0.

Al igual que una clase puede tener un constructor (en realidad puede tener más de uno, pero no adelantemos acontecimientos), también puede tener un destructor (en el caso del destructor sólo uno). En el destructor podemos añadir código que será ejecutado justo antes de que el objeto sea destruido de forma automática (sin necesidad de invocarlo), por ejemplo, por una salida fortuita del programa. Un destructor se declara igual que un constructor, pero anteponiendo el símbolo de tilde delante (~).

```
// Destructor de la clase coche
Coche::~Coche() {
   if (velocidad != 0)
      velocidad = 0;
}
```

Este destructor se asegura de que si el coche está en movimiento, su velocidad se reduzca a 0.

Herencia

No sé de color tienes los ojos, pero puedo asegurar que del mismo color que alguno de tus ascendientes. Este mecanismo biológico fue descrito por Mendel (armado con una buena dosis de paciencia y una gran cantidad de guisantes) y se llama herencia. La herencia se transmite de padres a hijos, nunca al revés. En POO, la herencia funciona igual, es decir, en un sólo sentido. Mediante la herencia, una clase hija (llamada subclase) hereda los atributos y los métodos de su clase padre. Vamos a verlo mejor con un ejemplo. Imaginemos que queremos crear una clase llamada `CochePolicia`, que además de acelerar y frenar pueda activar y desactivar una sirena. Podríamos crear una clase nueva llamada CochePolicia con los atributos y clases necesarios tanto para frenar y acelerar como para activar y desactivar la sirena. En lugar de eso, vamos a aprovechar que ya tenemos una clase llamada `Coche` y que ya contiene algunas de las funcionalidades que queremos incluir en `CochePolicia`. Veámoslo sobre un ejemplo.

```
Class CochePolicia:public Coche {

     private:
     int sirena;

     public:
     sirenaOn() { sirena=1; }
     sirenaOff() { sirena=0; }
}
```

Observa cómo hemos implementado los métodos `sirenaOn()` y `sirenaOff()` dentro de la misma clase. Esto es completamente válido, y se suele utilizar cuando el código del método es pequeño, como en este caso. Lo primero que nos llama la atención de la declaración de la clase es su primera línea. Tras el nombre de la clase, hemos añadido dos puntos seguidos de la clase padre, es decir, de la que heredamos los métodos y atributos. El modificador `public` hace que los métodos y atributos heredados mantengan sus modificadores de acceso originales. Si queremos que se hereden como protegidos, utilizaremos el modificador `protected`. Ahora la clase `CochePolicia` posee dos atributos, velocidad, que ha sido heredado y sirena, que ha sido declarado dentro de la clase `CochePolicia`. Con los métodos sucede exactamente igual. La clase hija ha heredado `acelerar()` y `frenar()`, y además le hemos añadido los métodos `sirenaOn()` y `sirenaOff()`. Un objeto instancia de CochePolicia pude utilizar sin ningún problema los métodos `acelerar()` y `frenar()` tal y como hacíamos con los objetos instanciados de la clase `Coche`.

Es posible heredar de dos o más clases a la vez. Esto se llama herencia múltiple. Para heredar de más de una clase, las enumeramos separadas por comas detrás de los dos puntos. Sinceramente, a no ser que te guste el riesgo, la herencia múltiple, si no se tiene cuidado, te traerá más dolores de cabeza que otra cosa.

Polimorfismo

El polimorfismo es otra de las grandes característica de la POO. La palabra polimorfismo deriva de poli (multiples) y del término griego morfos (forma). Es decir, múltiples formas.

Vamos a verlo con un ejemplo. Supongamos que queremos dotar al método frenar de más funcionalidad. Queremos que nos permita reducir hasta la velocidad que queramos. Para ello le pasaremos como parámetro la velocidad, pero también sería útil que frenara completamente si no le pasamos ningún parámetro. El siguiente código cumple estos requisitos.

```
// Declaración de la clase coche
class Coche {
   // Atributos de la clase coche
   int velocidad;

   // Métodos de la clase coche
   void acelerar(int velocidad);
   void frenar();
   void frenar(int velocidad);
};

void Coche::frenar() {
   // Ponemos a 0 el valor del atributo velocidad
   velocidad = 0;
}

void Coche::frenar() {
   // Ponemos a 0 el valor del atributo velocidad
   velocidad = 0;
}

void Coche::frenar(int velocidad) {
   // Reducimos la velocidad
   if (velocidad < this.velocidad)
      this.velocidad = velocidad;
}
```

Como ves tenemos dos métodos frenar. En C, esto causaría un error. Cuando llamemos al método `frenar()`, C++ sabrá cual tiene que ejecutar dependiendo de si lo llamamos con un parámetro de tipo entero o sin parámetros. Esto que hemos hecho se llama sobrecarga de métodos. Podemos crear tantas versiones diferentes del método siempre y cuando sean diferentes.

El constructor de una clase también puede ser sobrecargado.

Punteros y memoria

C++ añade un nuevo operador para reservar memoria y crear punteros. Se trata del operador `new`. Este operado reserva la memoria para contener un tipo de datos determinado, y devueve y puntero a la memoria reservada.

```
// Creamos un puntero de tipo coche
// Ojo, no creamos un objeto coche, sólo un puntero
coche *micoche;

// creamos un objeto coche y devolvemos
// un puntero al objeto
micoche = new Coche;

// Llamada a un método del objeto
micoche->acelera(100);
```

```
// liberar memoria
delete coche;
```

Lo primero que hemos hecho es declara un puntero de tipo coche. En este momento no hemos creado ningún objeto, sino un puntero. El encargado de crear el objeto es el operador `new`, que además de crear la instancia devuelve un puntero al objeto recién creado.

La siguiente línea llama al método acelera(), pero observa que en vez de un punto usamos los caracteres -> (signo de resta seguido del signo mayor que). Siempre que trabajamos con un puntero a un objeto sustituimos el punto por ->.

Por último, para liberar los recursos reservados por `new`, utilizamos el operador `delete`, que va a liberar la memoria ocupada por el objeto `Coche`.

Hemos visto en éste apéndice las características más importantes de C++ relacionadas con la POO, sin embargo este lenguaje es mucho más extenso y no podemos hacer una completa introducción por problemas de espacio. Si has sido capaz de asimilar la información de este apéndice me doy por contento por ahora, aunque mi consejo es que sigas profundizando en este lenguaje.

Apéndice

C

Recursos

Bibliografía

Programación

Programación orientada a objetos con C++. 2ª Edición.
Fco. Javier Ceballos
Ed. Ra-ma.
ISBN: 84-7897-268-4

Programacion en C.
Mitchell Waite/Stephen Prata
Ed. Anaya.
ISBN: 84-7614-374-5

Programación en Linux con ejemplos.
Kurt Wall
Ed. Prentice Hall.
ISBN: 987-9460-09-X

Visual C++ 6
Jon Bates/Tim Tomkins
Ed. Prentice Hall.
ISBN: 8483220776

Programación de videojuegos

Windows Game Programming for Dummies, Second Edition.
André LaMothe.
Ed. IDG Books.
ISBN: 0764516787

OpenGL Game Programming.
Dave Astle/kevin Hawkins/André LaMothe.
Ed. Learning Express
ISBN: 0761533303

AI Techniques for Game Programming.

Mat Buckland.
Ed. Learning Express
ISBN: 193184108X

Beginning Direct3D game programming, 2nd Edition.
Wolfgang f. Engel.
Ed. Learning Express
ISBN: 193184139X

Gráficos

Computer Graphics, principles and practice.
Foley/van Dam/Feiner/Hughes
Ed. Addison Wesley
ISBN: 0-201-12110-7

OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL, Version 1.2 (3rd Edition).
Mason Woo y otros
Ed. Addison-Wesley
ISBN: 0201604582

OpenGL SuperBible, Second Edition (2nd Edition).
Richard S. Wright Jr./Michael R. Sweet.
Ed. Waite Group Press
ISBN: 1571691642

Programación multimedia avanzada con DirectX
Constantino Sánchez Ballesteros
Ed. Ra-ma
ISBN: 8478973427

Enlaces

Programación

http://www.dcp.com.ar/

http://www.programacion.net/

http://www.planetiso.com/

http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html

Programación de videojuegos

http://www.flipcode.com/

http://www.gamedev.net/

http://www.gamasutra.com/

http://gpp.netfirms.com/

http://www.gametutorials.com/

http://www-cs-students.stanford.edu/~amitp/gameprog.html

Gráficos

http://nehe.gamedev.net/

http://www.opengl.org/

http://www.shaderx.com/direct3d.net/index.html

http://www.geocities.com/SiliconValley/Way/3390/index.html

Índice Alfabético

A

algoritmos de búsqueda, 106
alpha-blending, 32
atexit, 25
Audio, 51

B

bit blitting, 27

C

C++, 143
CD-ROM, 56

Ch

chunk, 70

C

clases, 144
códigos sym, 40
colisiones, 81

D

delete, 54, 149
Disparos, 108

E

enemigos, 117
event, 40
explosiones, 108

G

Game loop, 10
Gestión de eventos, 38

H

herencia, 147

I

IMG_Load, 66
inteligencia artificial, 105

L

linux, 141

M

mapas, 98
Mappy, 98
máquinas de estado, 107
Mix_AllocateChannels, 70
Mix_CloseAudio, 69
Mix_FadeInChannel, 71
Mix_FadeInChannelTimed, 71
Mix_FadeInMusic, 73
Mix_FadeOutChannel, 71
Mix_FadeOutMusic, 73
Mix_FreeChunk, 70
Mix_FreeMusic, 73
Mix_GetMusicType, 74
Mix_HaltChannel, 71
Mix_HaltMusic, 73
Mix_LoadMUS, 73
Mix_LoadWAV, 70
Mix_OpenAudio, 69
Mix_Pause, 71
Mix_Paused, 71
Mix_PausedMusic, 74
Mix_PauseMusic, 73
Mix_PlayChannel, 70
Mix_PlayChannelTimed, 70
Mix_Playing, 71
Mix_PlayingMusic, 74
Mix_PlayMusic, 73
Mix_Resume, 71
Mix_ResumeMusic, 73
Mix_RewindMusic, 73
Mix_SetMusicPosition, 73
Mix_VolumeChunk, 70
Mix_VolumeMusic, 73

N

new, 148
niveles, 119

O

objetos, 144

P

Pantalla inicial, 121
polimorfismo, 147
puntación, 121
punteros, 148

R

Ratón, 39
red neuronal, 105

S

Scrolling, 100
SDL, 23
SDL_ActiveEvent, 46
SDL_AudioSpec, 51
SDL_BlitSurface, 30
SDL_BUTTON, 44
SDL_CD, 56
SDL_CDClose, 57
SDL_CDEject, 57
SDL_CDName, 56
SDL_CDNumDrives, 56
SDL_CDOpen, 57
SDL_CDPause, 57
SDL_CDPlay, 57
SDL_CDPlayTracks, 57
SDL_CDResume, 57
SDL_CDStatus, 57
SDL_CDStop, 57
SDL_CDtrack, 56
SDL_CloseAudio, 52

SDL_ConvertSurface, 37
SDL_CreateRGBSurface, 27
SDL_Delay, 61
SDL_EnableUNICODE, 43
SDL_ExposeEvent, 46
SDL_FillRect, 31
SDL_Flip, 31
SDL_FreeSurface, 31
SDL_FreeWAV, 52
SDL_GetClipRect, 37
SDL_GetError, 25
SDL_GetKeyState, 43
SDL_GetTicks, 60
SDL_GetVideoInfo, 38
SDL_image, 66
SDL_Init, 24
SDL_InitSubSystem, 25
SDL_JoyAxisEvent, 45
SDL_JoyButtonEvent, 45
SDL_JoystickClose, 47
SDL_JoystickEventState, 45
SDL_JoystickGetAxis, 48
SDL_JoystickGetButton, 48
SDL_JoystickName, 47
SDL_JoystickNumAxes, 48
SDL_JoystickNumButtons, 48
SDL_JoystickOpen, 47
SDL_JoystickOpened, 48
SDL_JoystickUpdate, 48
SDL_KeyboardEvent, 40
SDL_keysym, 40
SDL_LoadBMP, 30
SDL_LoadWAV, 52
SDL_MapRGB, 31, 33
SDL_mixer, 69
SDL_MouseButtonEvent, 44
SDL_MouseMotionEvent, 44
SDL_NumJoysticks, 47
SDL_OpenAudio, 52
SDL_PauseAudio, 52
SDL_PixelFormat, 27
SDL_PollEvent, 40
SDL_Quit, 25
SDL_QuitEvent, 45
SDL_QuitSubSystem, 25
SDL_Rect, 30
SDL_ResizeEvent, 46
SDL_SetAlpha, 36
SDL_SetClipRect, 36
SDL_SetColorKey, 32
SDL_SetVideoMode, 26
SDL_Surface, 30
SDL_ttf, 62
SDL_UpdateRect, 31
SDL_VideoInfo, 38
SDL_WM_GetCaption, 60
SDL_WM_IconifyWindow, 60
SDL_WM_SetCaption, 60
SDL_WM_SetIcon, 60
sistemas basados en reglas, 106
sprites, 76

T

Teclado, 39
temporización, 120
this, 145
tiles, 93
Timming, 60
Transparencias, 32
TTF_Close, 63
TTF_Font, 63
TTF_GetFontStyle, 63
TTF_Init, 62
TTF_OpenFont, 62
TTF_OpenFontIndex, 62
TTF_Quit, 62
TTF_RenderText_Blended, 63
TTF_RenderText_Shaded, 63
TTF_RenderText_Solid, 63
TTF_SetFontStyle, 63

V

vidas, 121
Video, 26
Visual C++, 137

W

Window Manager, 60