Tao.Sdl paso a paso

0. Introducción

Este documento es un tutorial donde, partiendo de los conceptos explicados en los apuntes y anexos de programación del profesor Nacho Cabanes para el módulo de Programación del IES San Vicente, se pretende dar una visión más guiada, explicando por separado cada uno de los conceptos fundamentales de uso de la librería TAO SDL allí explicada. El objetivo es simplemente dar una visión más guiada del proceso, para poderla abarcar paso a paso desde una formación semipresencial.

Agradecimientos

Antes de comenzar, es necesario decir que este tutorial ha surgido gracias, en primer lugar, a todos los materiales y ayuda proporcionada por Nacho Cabanes, tanto de forma directa, como en los apuntes que se utilizan para la asignatura de Programación en el IES San Vicente, como a través de los muchos recursos y tutoriales disponibles en su web (*nachocabanes.com*).

En segundo lugar, gracias a Mario Vivas por sus puntualizaciones y ayuda en algunos apartados en los que él ha estado investigando por su cuenta, y me ha ido contando las cosas que ha visto. Especialmente, en el tratamiento de audio, fuentes, y la gestión de eventos con Tao.Sdl.

¿Qué es SDL?

Como se comenta en los apuntes de Nacho Cabanes, SDL es una librería para realizar videojuegos en diversas plataformas, que permite tanto el dibujado de imágenes en pantalla como el control de periféricos (teclado, ratón, gamepad), entre otras cosas. Funciona de forma nativa para C++, y se han realizado extensiones para poder utilizarla en otros lenguajes, como Python o C#. En este tutorial veremos los pasos básicos para utilizarla en este último lenguaje.

La librería Tao.Sdl para C#

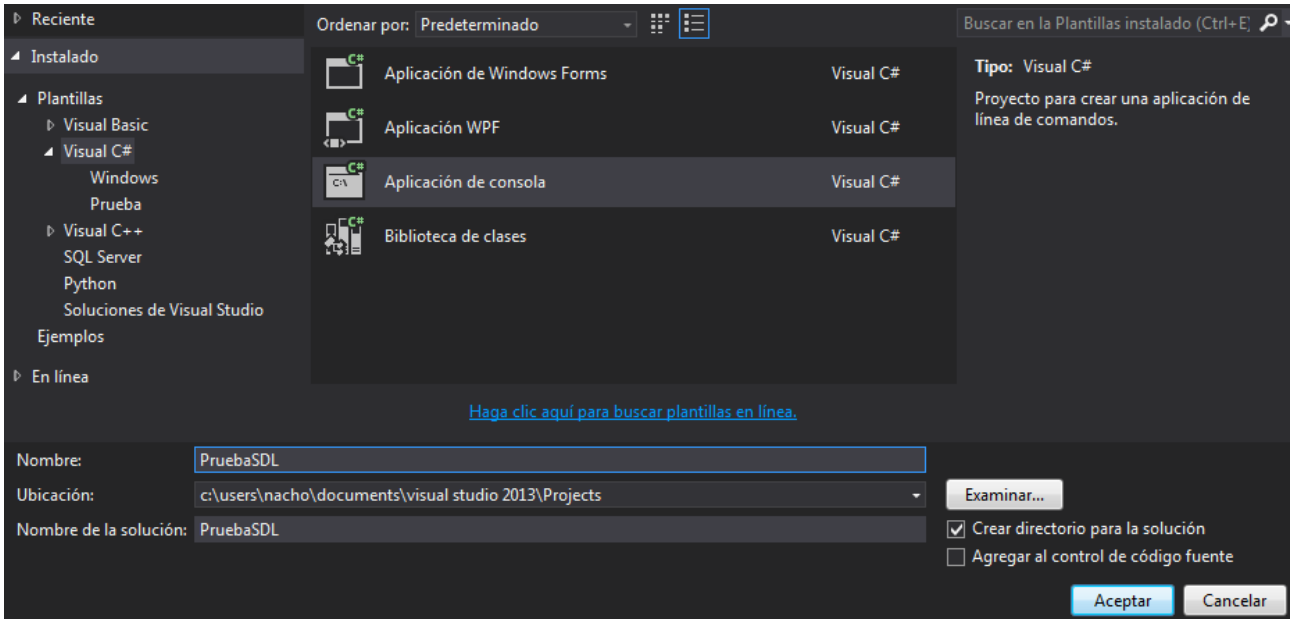
En lo relativo a C#, se dispone de la extensión Tao.Sdl, que es una adaptación de la librería original para poderla utilizar en este lenguaje. Inicialmente formaba parte del proyecto mono (el compilador gratuito para C#), pero se abandonó, y las últimas versiones forman parte de un proyecto SourceForge que podemos consultar en su web oficial: *http://sourceforge.net/projects/taoframework/*

Descarga e instalación de Tao.Sdl

Para descargar Tao.Sdl, por tanto, accederemos a la URL anterior del proyecto SourceForge, descargamos el instalador y lo instalamos con las opciones por defecto. Se habrá creado una carpeta *TaoFramework* en nuestra carpeta de *Archivos de programa (x86)*. Es importante tener esta carpeta localizada, porque la necesitaremos para copiar ciertas librerías en nuestros proyectos.

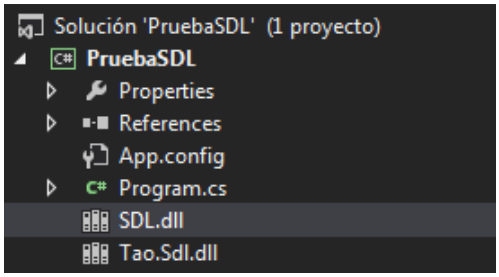
1. Crear y configurar el proyecto

Una vez instalada Tao.Sdl, ya podemos crear un proyecto (en VisualStudio o SharpDevelop) que la utilice. En este tutorial vamos a crear un proyecto llamado **PruebaSDL**, que inicialmente será una aplicación de consola.

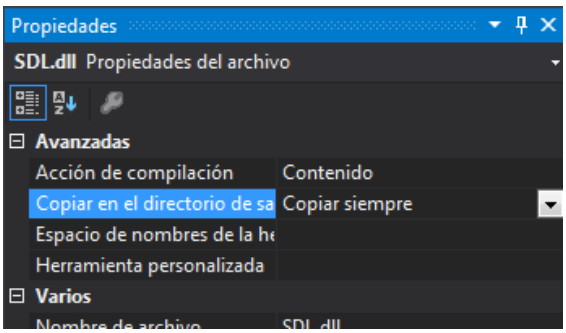


Una vez creado el proyecto, debemos localizar en la carpeta de instalación de Tao.Sdl (carpeta *TaoFramework* comentada antes) un par de librerías DLL, y copiarlas a la carpeta de fuentes de nuestro proyecto. Estas librerías son:

- Librería *SDL.dll* dentro de la subcarpeta *lib* de *TaoFramework*.
- Librería *Tao.Sdl.dll*, dentro de la subcarpeta *bin* de *TaoFramework*.



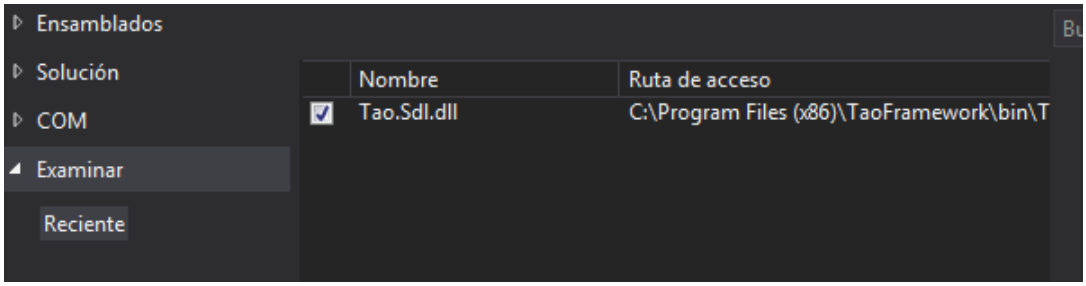
Además, indicaremos en las propiedades de estos ficheros que se copien siempre a la carpeta de salida, para poder tener disponibles las DLL en la carpeta *Debug* cuando compilemos y/o ejecutemos el programa.



Además, es bastante probable que para la gran mayoría de nuestros proyectos en Tao.Sdl necesitemos otras librerías DLL que podemos localizar en la subcarpeta *lib* de la instalación de Tao. Podemos copiarlas y pegarlas en el proyecto, y configurarlas de la misma forma que las dos anteriores, para así dejarnos ya ese trabajo hecho. Estas librerías son:

- *SDL_image.dll* para cargar imágenes en el juego
- *SDL_ttf.dll* y *libfreetype-6.dll* para mostrar textos con fuentes TTF en los juegos
- *SDL_mixer.dll* para poder añadir audios con música y efectos al juego
- *zlib1.dll*, una librería auxiliar que en ocasiones es utilizada por otras

Finalmente, en las referencias del proyecto, deberemos añadir la librería *Tao.Sdl.dll* para poder utilizar las clases que hay en ella. Hacemos clic derecho en las referencias (*References*) y elegimos *Agregar referencia*. Buscamos la DLL mediante el botón de *Examinar* y marcamos su casilla.



Si posteriormente necesitamos alguna librería adicional más (por ejemplo, para trabajar con imágenes PNG o JPG), las copiaremos en el proyecto siguiendo este mismo procedimiento (añadir la DLL a la carpeta de fuentes y decirle que se copie siempre a la carpeta de salida). Veremos el caso de alguna de estas librerías en secciones posteriores.

2. Configurar la pantalla o ventana

Una vez creado el proyecto y configuradas las librerías, lo primero que vamos a hacer es una pequeña versión del juego que simplemente configure la resolución de pantalla, y permita ejecutar el juego a pantalla completa o embebido en una ventana.

Crear y programar la clase Hardware

Para controlar todo lo relativo al hardware (comunicación con la pantalla, recogida de eventos de teclado, etc.), vamos a crear una clase *Hardware*, que de alguna forma encapsule los métodos de SDL en otros propios, que sean más sencillos de utilizar en nuestra aplicación ya que, como veremos, los métodos de SDL son a veces poco intuitivos en cuanto a lo que hacen, o a los parámetros que necesitan.

```
using System;
using Tao.Sdl;

namespace PruebaSDL
{
    class Hardware
    {
    }
}
```

Dentro de esta clase *Hardware*, vamos a definir unos atributos para almacenar la anchura y altura de pantalla, y la profundidad de color en bits. Además, definiremos un atributo de tipo *IntPtr* para gestionar la pantalla en sí (veremos que este tipo de datos se utiliza para referenciar o apuntar a elementos cualesquiera, y lo usaremos para apuntar a imágenes, audios, etc.)

```
class Hardware
{
    // Características de vídeo: anchura, altura y profundidad de color
    short anchoPantalla;
    short altoPantalla;
    short bitsColor;
    IntPtr pantalla;
```

Definimos también un constructor que dé valor a estos atributos, e inicialice la pantalla con esas características de resolución (ancho y alto) y profundidad de color:

```
public Hardware(short ancho, short alto, short bits, bool pantallaCompleta)
{
    anchoPantalla = ancho;
    altoPantalla = alto;
    bitsColor = bits;

    // Flags para el modo de pantalla
    int flags = Sdl.SDL_HWSURFACE | Sdl.SDL_DOUBLEBUF | Sdl.SDL_ANYFORMAT;
    if (pantallaCompleta)
       flags = flags | Sdl.SDL_FULLSCREEN;

    // Inicializar SDL
    Sdl.SDL_Init(Sdl.SDL_INIT_EVERYTHING);
    pantalla = Sdl.SDL_SetVideoMode(anchoPantalla, altoPantalla, bitsColor, flags);
    // Características del modo gráfico (ancho, alto, profundidad y otros flags)
    Sdl.SDL_Rect rect = new Sdl.SDL_Rect(0, 0, anchoPantalla, altoPantalla);
    // Rectángulo para recortar lo que quede fuera de la pantalla
```

```
    Sdl.SDL_SetClipRect(pantalla, ref rect);
}
```

El cuarto parámetro del constructor nos servirá para poder iniciar el juego indistintamente a pantalla completa (*true*) o embebido en una ventana (*false*). Notar que usamos el atributo *pantalla* para dibujar en él el rectángulo que va a suponer la pantalla del juego.

También podemos definir un destructor para eliminar los datos de SDL al cerrar el juego:

```
~Hardware()
{
    Sdl.SDL_Quit();
}
```

Probando la configuración

Para probar cómo funciona todo esto, vamos al *Main* de nuestro proyecto, y añadimos estas líneas para crear un nuevo objeto de tipo *Hardware* y mostrarlo:

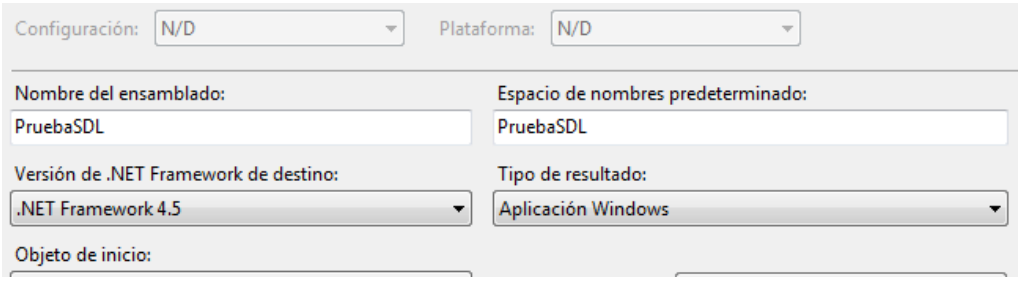
```
using System;
using System.Threading;

namespace PruebaSDL
{
    class Program
    {
        static void Main(string[] args)
        {
            Hardware h = new Hardware(800, 600, 24, false);
            Thread.Sleep(5000);
        }
    }
}
```

La última instrucción (*Thread.Sleep*, para la que necesitaremos añadir el *using* de *System.Threading*) no será realmente necesaria en un futuro, o al menos no de esta forma, pero ahora, como nuestro juego no hace nada, si no la pusiéramos veríamos cómo se abre y cierra la ventana instantáneamente. Así, tras abrir la ventana del juego, esperará 5 segundos, y la cerrará. También podemos comprobar que, cambiando el último parámetro del constructor de *Hardware* a *true*, tendremos pantalla completa.

Cambiar el tipo de aplicación

Ahora que ya hemos comprobado que el videojuego "funciona" (es decir, muestra la pantalla en negro), podemos cambiar el tipo de aplicación. Al crear el proyecto, la habíamos definido de tipo *Aplicación de consola*, ya que de lo contrario nos habría obligado a definir una ventana innecesariamente. Pero de esta forma, se nos abrirán dos ventanas al ejecutar: una el juego, y la otra una consola.

Para evitar la ventana de la consola tenemos que cambiar el tipo de aplicación: hacemos clic derecho sobre el nombre del proyecto, vamos a *Propiedades*, y donde pone *Tipo de resultado* cambiamos *Aplicación de consola* por *Aplicación Windows*:


Configuración: N/D
Plataforma: N/D
Nombre del ensamblado:
PruebaSDL
Espacio de nombres predeterminado:
PruebaSDL
Versión de .NET Framework de destino:
.NET Framework 4.5
Tipo de resultado:
Aplicación Windows
Objeto de inicio:

3. Dibujar una imagen

Vamos a ver en este paso cómo mostrar una imagen en la escena del videojuego. Para poder trabajar con imágenes de varios tipos, necesitamos añadir a nuestro proyecto la librería *SDL_image.dll*, como hemos explicado en la sección 1.

Crear y programar la clase Imagen

Además, y de nuevo para simplificar el manejo de las clases y métodos ofrecidos por Tao.Sdl, vamos a crear nuestra propia clase, llamada Imagen.

```
using System;
using Tao.Sdl;

namespace PruebaSDL
{
    class Imagen
    {
    }
}
```

En ella, vamos a definir como atributos las coordenadas X e Y de la esquina superior izquierda donde queramos mostrar la imagen, y la anchura y altura que va a tener la imagen en pantalla. Definiremos también un objeto de tipo *IntPtr*, que en este caso nos va a servir como referencia a la imagen con la que vamos a trabajar.

```
class Imagen
{
    // Referencia a la imagen con que trabajar
    IntPtr imagen;
    // Coordenadas X e Y de la esquina superior izquierda donde poner la imagen
    short x, y;
    // Anchura y altura de la imagen en la pantalla
    short ancho, alto;
```

Definiremos también un constructor que reciba como parámetros el nombre del fichero (imagen) a cargar, y el ancho y alto que tiene la imagen, y con el que se mostrará en pantalla (atributos *ancho* y *alto*).

```
public Imagen(string nombreFichero, short ancho, short alto)
{
    imagen = SdlImage.IMG_Load(nombreFichero);
    if (imagen == IntPtr.Zero)
    {
        System.Console.WriteLine("Imagen inexistente!");
        Environment.Exit(1);
    }
    this.ancho = ancho;
    this.alto = alto;
}
```

El método *SdlImage.IMG_Load* en principio permite cargar cualquier tipo de imagen a elegir entre diversos formatos (BMP, JPG, PNG, TIFF, etc.), pero para algunos de estos formatos (por ejemplo, PNG o JPG), veremos a continuación que necesitamos tener añadidas algunas librerías adicionales a nuestro proyecto.

Siguiendo con nuestra clase *Imagen*, y por comodidad para futuras ampliaciones, también definiremos un método llamado *MoverA*, que recibirá como parámetros las

coordenadas X e Y donde situar la esquina superior izquierda de la imagen, y las asignará a los atributos *x* e *y*:

```
public void MoverA(short x, short y)
{
    this.x = x;
    this.y = y;
}
```

Finalmente, añadimos los *getters* correspondientes a la clase, para poder acceder a los atributos desde fuera de la clase:

```
public IntPtr GetPuntero() { return imagen; }
public short GetX() { return x; }
public short GetY() { return y; }
public short GetAncho() { return ancho; }
public short GetAlto() { return alto; }
```

Cambios en la clase Hardware

Para poder volcar esta imagen a la pantalla, debemos añadir un método nuevo a nuestra clase *Hardware* que se encargue de dibujar una imagen

```
public void DibujarImagen(Imagen imagen)
{
      Sdl.SDL_Rect origen = new Sdl.SDL_Rect(0, 0, imagen.GetAncho(), imagen.GetAlto());
      Sdl.SDL_Rect dest = new Sdl.SDL_Rect(imagen.GetX(), imagen.GetY(),
         imagen.GetAncho(), imagen.GetAlto());
      Sdl.SDL_BlitSurface(imagen.GetPuntero(), ref origen, pantalla, ref dest);
}
```

En la variable *origen* definiremos el rectángulo de la imagen original que queremos tomar para mostrar en la pantalla. En nuestro caso, por simplificar, vamos a elegir toda la imagen, desde el inicio (0,0) hasta el punto final (imagen.GetAncho(), imagen.GetAlto()).

En la variable *dest* indicamos el rectángulo de la pantalla donde vamos a dibujar la imagen (coordenadas X e Y de la imagen, y a partir de ahí tomamos la anchura y altura de la imagen para formar el rectángulo).

El método *SDL_BlitSurface* lo usaremos para dibujar la imagen origen (*imagen.GetPuntero()*) en la pantalla, tomando de la primera el rectángulo *origen* especificado, y volcándolo en el rectángulo *destino* indicado para la segunda.

Además, como lo que dibujamos lo estamos dibujando en una pantalla que no se muestra directamente, debemos añadir un método más a *Hardware* para hacer visible la *pantalla*.

```
public void VisualizarPantalla()
{
    Sdl.SDL_Flip(pantalla);
}
```

Probar los cambios

Para probar estos cambios en el programa, vamos a nuestro bloque *Main* y añadimos las líneas correspondientes a cargar una imagen. Antes de eso, busca una imagen (de momento BMP para no tener problemas), con algún personaje que te guste, preferiblemente con el fondo negro para que no desentone, y añádela a la carpeta de fuentes del proyecto, indicando que se copie siempre a la carpeta de salida.

Después, dejamos el método *Main* aproximadamente así (el nombre de la imagen dependerá del archivo que hayamos copiado al proyecto, y el ancho, alto, X e Y pueden ser los que nosotros queramos, teniendo en cuenta la resolución que hemos dicho en la pantalla (800 x 600 en este ejemplo) y el tamaño de la imagen elegida:

```
static void Main(string[] args)
{
     Hardware h = new Hardware(800, 600, 24, false);
     Imagen img = new Imagen("personaje.bmp", 50, 50);
     img.MoverA(100, 200);
     h.DibujarImagen(img);
     h.VisualizarPantalla();
     Thread.Sleep(5000);
}
```

Cargar otros formatos de imagen

Si queremos trabajar con otros formatos de imagen, como por ejemplo PNG o JPG, necesitamos añadir más librerías DLL a nuestro proyecto:

- En el caso de imágenes PNG, necesitaremos añadir las librerías *libpng12.dll* y *libpng12-0.dll*
- En el caso de imágenes JPG, necesitaremos la librería *jpeg.dll*
- En ambos casos necesitaremos, además, la librería *zlib1.dll*, que ya hemos indicado en la sección 1 de este tutorial que estaría bien dejar añadida por defecto.

Estas librerías están en la subcarpeta *lib* de la instalación de Tao.Sdl. Deberemos copiarlas a la carpeta de fuentes y hacer que se copien siempre a la carpeta de salida, como en las librerías anteriores.

Escalar una imagen

La forma que hemos visto antes de mostrar una imagen en pantalla sirve para mostrarla tal cual, con su tamaño real. Es posible que en ocasiones nos interese mostrarla con un tamaño más pequeño. Por ejemplo, porque la imagen que hemos encontrado es algo grande para lo que lo queremos usar. En ese caso, la mejor alternativa es escalar la imagen con algún programa de edición de imagen, como por ejemplo *Gimp*, y dejarla con el tamaño exacto que queramos que tenga.

> Por ejemplo, en el caso anterior, hemos indicado que queremos una imagen de 50 píxeles de ancho y 50 de alto. Si la imagen que hemos elegido no tiene ese tamaño, deberíamos escalarla previamente.

Otra alternativa, que no todas las librerías de juegos tienen, y que a veces no es muy aconsejable, es utilizar las funciones de la librería para escalar "en caliente" la imagen cuando la vayamos a dibujar o cargar en el programa. Esto tiene el inconveniente del rendimiento, ya que si cada vez que vamos a dibujar la imagen tenemos que cambiarle el tamaño, la velocidad de carga de cada frame del juego puede verse afectada.

4. Responder al teclado

Vamos a hacer ahora que la imagen que hemos añadido en el paso anterior responda a nuestras pulsaciones de teclado y, por ejemplo, se mueva en las cuatro direcciones de las cuatro flechas del cursor. También aprovecharemos para quitar la espera de 5 segundos, y hacer directamente que el videojuego termine cuando pulsemos la tecla *Escape*, por ejemplo.

Cambios en la clase Hardware: eventos de teclado

En primer lugar, vamos a añadir a nuestra clase Hardware una serie de constantes que correspondan con las constantes definidas en Tao.Sdl para cada tecla que podemos pulsar. En realidad, no vamos a definir todas las posibles teclas, sólo las que vayamos a necesitar (en este caso, los cursores y la tecla de Escape):

```
public static int TECLA_ESC = Sdl.SDLK_ESCAPE;
public static int TECLA_ARR = Sdl.SDLK_UP;
public static int TECLA_ABA = Sdl.SDLK_DOWN;
public static int TECLA_IZQ = Sdl.SDLK_LEFT;
public static int TECLA_DER = Sdl.SDLK_RIGHT;
```

Después, añadimos un método que recoja la pulsación de una tecla, y devuelva qué tecla de todas las contempladas se ha pulsado:

```
public bool TeclaPulsada(int c)
{
    bool pulsada = false;
    Sdl.SDL_PumpEvents();
    Sdl.SDL_Event suceso;
    Sdl.SDL_PollEvent(out suceso);
    int numkeys;
    byte[] teclas = Sdl.SDL_GetKeyState(out numkeys);
    if (teclas[c] == 1)
        pulsada = true;
    return pulsada;
}
```

Probar los cambios

En el programa principal, definiremos ahora un bucle donde recogeremos la tecla que pulse el usuario, y moveremos al personaje en la dirección indicada (cambiando las coordenadas X e Y de la imagen), hasta que pulsemos Escape. Mantenemos el *using* de *System.Threading*, y ahora haremos que el programa duerma 20 ms tras cada iteración, para no acaparar todo el tiempo de procesador.

```
static void Main(string[] args)
{
    bool terminado = false;
    short x = 100, y = 200;
    Hardware h = new Hardware(800, 600, 24, false);
    Imagen img = new Imagen("personaje.bmp", 50, 50);

    do
    {
        if (h.TeclaPulsada(Hardware.TECLA_ARR))
            y -= 2;
        if (h.TeclaPulsada(Hardware.TECLA_ABA))
```

```
            y += 2;
        if (h.TeclaPulsada(Hardware.TECLA_IZQ))
            x -= 2;
        if (h.TeclaPulsada(Hardware.TECLA_DER))
            x += 2;
        if (h.TeclaPulsada(Hardware.TECLA_ESC))
            terminado = true;
        img.MoverA(x, y);

        h.DibujarImagen(img);
        h.VisualizarPantalla();
        Thread.Sleep(20);
    } while (!terminado);
}
```

Más cambios en la clase Hardware y el Main

Cuando probemos el juego tal cual está ahora mismo, veremos que, a medida que movemos el personaje por la pantalla, va dejando un rastro tras de sí. Eso es porque deberíamos borrar la pantalla antes de volver a dibujar la nueva posición del personaje, para que no se quede el rastro de donde estaba antes.

Para ello, debemos añadir un método en nuestra clase *Hardware* que se encargue de limpiar la pantalla cuando lo necesitemos:

```
public void BorrarPantalla()
{
    Sdl.SDL_Rect origen = new Sdl.SDL_Rect(0, 0, anchoPantalla, altoPantalla);
    Sdl.SDL_FillRect(pantalla, ref origen, 0);
}
```

El método *SDL_FillRect* sirve para dibujar un rectángulo. En este caso lo dibujaremos en la pantalla, con el tamaño del rectángulo definido en *origen*, y con el color del tercer parámetro (0, que equivale al negro).

Además, en el *Main*, haremos que se llame a este método dentro del bucle, antes de redibujar y mostrar la pantalla.

```
        ...
        h.BorrarPantalla();
        h.DibujarImagen(img);
        h.VisualizarPantalla();
        Thread.Sleep(20);
    } while (!terminado);
}
```

5. Animar las imágenes. Sprites

Ya sabemos mostrar imágenes en pantalla, y hacer que algunas se muevan con nuestras acciones de teclado. Vamos a aprender ahora que, en ese movimiento, o bien sin que pulsemos nada, la imagen cambie para hacer un movimiento animado, simulando, por ejemplo, que el personaje camina. Para conseguir esto, necesitamos tener varias imágenes del personaje u objeto, cada una con una posición diferente. A cada una de esas imágenes se le suele llamar **sprite**, y correspondería a un paso de la animación a realizar. Esto se puede hacer de dos formas diferentes:

1. Teniendo cada imagen en un archivo separado (poco útil si hay muchos sprites para una animación y/o muchos personajes que animar)
2. Teniendo una hoja de sprites (*sprite sheet*) para cada personaje que queramos animar. Estas hojas vienen con un conjunto de sprites de ese personaje, de forma que podemos elegir qué sprite de esa hoja nos interesa tomar como imagen en cada momento, definiendo el rectángulo que lo engloba.

Veremos estas dos formas de utilizar sprites a continuación

Animación mediante imágenes separadas

Si, por ejemplo, quisiéramos hacer una animación sobre Pacman moviéndose hacia la derecha, tendríamos que contar, por ejemplo, con estos tres sprites (en tres imágenes separadas):

Y, en nuestro código, cada vez que pulsemos la tecla de movimiento a la derecha, tendríamos que ir alternando una de estas imágenes, formando una secuencia. Para ello, podemos guardar las 3 imágenes en un array (los nombres de archivos, o bien tres objetos de tipo *Imagen* con la imagen ya precargada), y cambiar de una a otra con cada pulsación de la flecha derecha:

```
static void Main(string[] args)
{
    bool terminado = false;
    Imagen[] imagenes = new Imagen[3];
    imagenes[0] = new Imagen("pacman.bmp", 50, 50);
    imagenes[1] = new Imagen("pacman_der_1.bmp", 50, 50);
    imagenes[2] = new Imagen("pacman_der_2.bmp", 50, 50);
    short x = 100, y = 200;
    Hardware h = new Hardware(800, 600, 24, false);
    int i = 0;

    do
    {
        if (h.TeclaPulsada(Hardware.TECLA_DER))
        {
            x += 2;
            i = (i + 1) % imagenes.Length;
        }
        if (h.TeclaPulsada(Hardware.TECLA_ESC))
```

```
            terminado = true;
        imagenes[i].MoverA(x, y);

        h.BorrarPantalla();
        h.DibujarImagen(imagenes[i]);
        h.VisualizarPantalla();
        Thread.Sleep(20);
    } while (!terminado);
}
```

Previamente, según este código, tendríamos que haber añadido estas imágenes a la carpeta del proyecto, para que se copien a la carpeta de salida, si queremos que nos funcionen con las rutas que hemos indicado en dicho código.

De la misma forma, podríamos conseguir los sprites para el movimiento en otras direcciones, y mover al personaje en esas direcciones.

Animación mediante hojas de sprites

La segunda forma de conseguir animaciones de personajes es unir todos los sprites de un personaje en una sola imagen. Para ello podemos valernos de programas especializados y de pago como *TexturePacker*, o bien utilizar editores de imágenes gratuitos como Gimp, y montarnos nosotros la hoja de sprites.

En nuestro caso, y volviendo al ejemplo de Pacman anterior, tendríamos una imagen como la siguiente:

donde unimos los tres sprites de tamaño 50 x 50, de forma que, recortando la parte de la imagen que nos interese, podemos mostrar en cada momento del juego una u otra franja recortada.

Para ello, añadiremos un nuevo método *DibujarImagen* a nuestra clase *Hardware* en el que, además de indicar la imagen a dibujar, indiquemos qué rectángulo de esa imagen queremos dibujar, representado por las coordenadas x e y de su esquina superior izquierda, y la anchura y altura a recortar desde ese punto.

```
public void DibujarImagen(Imagen imagen, short x, short y, short ancho, short alto)
{
    Sdl.SDL_Rect origen = new Sdl.SDL_Rect(x, y, ancho, alto);
    Sdl.SDL_Rect dest = new Sdl.SDL_Rect(imagen.GetX(), imagen.GetY(), ancho, alto);
    Sdl.SDL_BlitSurface(imagen.GetPuntero(), ref origen, pantalla, ref dest);
}
```

En el programa principal, tendríamos una variable de tipo *Imagen* (en lugar de un array, como en el caso anterior), que almacenará la hoja de sprites, y un array de coordenadas de cada una de las imágenes a recortar de la hoja de sprites, junto con la anchura y altura (opcionalmente, en el caso de que los sprites no sean todos del mismo tamaño).

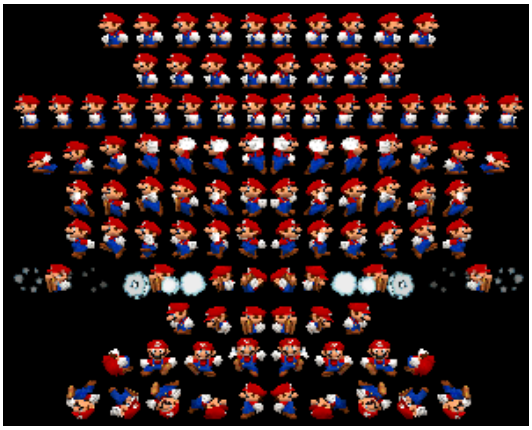
En nuestro caso, como todos los sprites miden 50 x 50, basta con que pongamos las coordenadas X e Y de inicio de cada sprite:

```
static void Main(string[] args)
{
    bool terminado = false;
    Imagen imagen = new Imagen("pacman_sprites.bmp", 50, 50);;
    short[,] coordenadas = { { 0, 0 }, { 50, 0 }, { 100, 0 } };
    short x = 100, y = 200;
    Hardware h = new Hardware(800, 600, 24, false);
    int i = 0;

    do
    {
        if (h.TeclaPulsada(Hardware.TECLA_DER))
        {
            x += 2;
            i = (i + 1) % coordenadas.GetLength(0);
        }
        if (h.TeclaPulsada(Hardware.TECLA_ESC))
            terminado = true;
        imagen.MoverA(x, y);

        h.BorrarPantalla();
        h.DibujarImagen(imagen, coordenadas[i,0], coordenadas[i,1], 50, 50);
        h.VisualizarPantalla();
        Thread.Sleep(20);
    } while (!terminado);
}
```

Creación u obtención de sprites

La creación u obtención de sprites variados para conseguir animar a un personaje puede ser una tarea pesada. En el caso de personajes sencillos como Pacman lo podemos hacer nosotros mismos con programas como Paint, Inkscape o similares, creando una imagen de tamaño determinado (por ejemplo, de 50 x 50), y hacer en cada imagen de ese tamaño una posición diferente de Pacman (con la boca más o menos abierta).

Para personajes con un dibujo más elaborado, es más complicado poder hacer ese dibujo nosotros mismos. En ese caso (y siempre que el videojuego no sea para publicar o para un uso comercial), podemos acudir a páginas de sprites ya hechos de personajes conocidos de videojuegos (Super Mario, Sonic, etc.), y recortar los sprites que nos interesen para nuestro propósito. Por ejemplo, si buscamos "*sprites super mario*" en Google podemos encontrar hojas de sprites como esta:



Y "sólo" tendríamos que, utilizando esa hoja, recortar el sprite que nos interese en cada momento de la animación, sabiendo sus coordenadas X e Y (de la esquina superior izquierda), y la anchura y altura del sprite.

Control del tiempo para el cambio de sprites

El simple hecho de cambiar de un sprite a otro tras cada pulsación de teclado o iteración del bucle puede no producir el efecto deseado, al resultar una animación demasiado rápida.

Una alternativa es hacer que se cambie de sprite tras un cierto período de tiempo, o con una determinada frecuencia. Hay diferentes formas de conseguir esto, pero quizá una forma sencilla sea contar cuántas veces se ha repetido el bucle principal, y cambiar de imagen cada X repeticiones. O bien, ya que en ocasiones el cambio depende del movimiento con el teclado, contar cuántas veces se ha pulsado la tecla correspondiente, y cambiar de imagen cada X pulsaciones de tecla. Si por ejemplo queremos cambiar de imagen cada 3 pulsaciones en el ejemplo anterior, tendríamos algo como esto:

```
...
int i = 0;
int contPulsaciones = 0;

do
{
    if (h.TeclaPulsada(Hardware.TECLA_DER))
    {
        x += 2;
        contPulsaciones++;
        if (contPulsaciones % REPETICIONES_SPRITE == 0)
        {
            contPulsaciones = 0;
            i = (i + 1) % coordenadas.GetLength(0);
        }
    }
...
```

siendo REPETICIONES_SPRITE una constante con el valor que queramos (en este caso, 3).

6. Mostrar textos

En la gran mayoría de juegos 2D que podamos hacer necesitaremos mostrar información por pantalla. Por ejemplo, un contador de puntos que llevemos en una pantalla del juego, títulos de crédito, etc.

Para trabajar con textos en Tao.Sdl, y suponiendo que vamos a trabajar con fuentes TTF que son las más habituales, necesitamos añadir las librerías *SDL_ttf.dll* y *libfreetype-6.dll* como ya comentamos en la sección 1. También necesitaremos añadir a nuestro proyecto los archivos TTF de las fuentes que queramos utilizar. Podemos encontrar algunas preinstaladas, por ejemplo, en la carpeta *Windows/Fonts*, o descargar otras en webs que recopilan diversas fuentes, como *www.fontsquirrel.com*.

Creación de la clase Fuente

Una vez más, para encapsular algunos de los métodos ofrecidos por Tao.Sdl para manipular fuentes, vamos a crearnos nuestra propia clase *Fuente*:

```
using System;
using Tao.Sdl;

namespace PruebaSDL
{
    class Fuente
    {
    }
}
```

Añadimos un atributo de tipo *IntPtr* que va a apuntar al tipo de letra que vamos a crear. Definimos un constructor al que le pasamos el nombre del archivo con la fuente (el archivo TTF), y el tamaño al que queremos mostrar la fuente, y creará con ellos el tipo de letra. Definimos también un *getter* para obtener ese tipo de letra.

```
IntPtr tipoDeLetra;

public Fuente(string nombreFichero, int tamano)
{
    tipoDeLetra = SdlTtf.TTF_OpenFont(nombreFichero, tamanyo);
    if (tipoDeLetra == IntPtr.Zero)
    {
        System.Console.WriteLine("Tipo de letra inexistente!");
        Environment.Exit(2);
    }
}

public IntPtr GetPuntero()
{
    return tipoDeLetra;
}
```

Cambios en la clase Hardware

En la clase Hardware, añadimos un nuevo método que dibuje un texto en pantalla, en unas coordenadas X e Y específicas, con un color RGB específico (indicando por separado las componentes R, G y B) y con la fuente que digamos. Lo que hará será renderizar el

texto indicado para transformarlo en una imagen, y plasmar la imagen en pantalla como una imagen cualquiera.

```
public void EscribirTexto(string texto, short x, short y,
                                byte r, byte g, byte b, Fuente fuente)
{
    Sdl.SDL_Color color = new Sdl.SDL_Color(r, g, b);
    IntPtr textoComoImagen = SdlTtf.TTF_RenderText_Solid(fuente.GetPuntero(), texto, color);
    if (textoComoImagen == IntPtr.Zero)
        Environment.Exit(5);
    Sdl.SDL_Rect origen = new Sdl.SDL_Rect(0, 0, anchoPantalla, altoPantalla);
    Sdl.SDL_Rect dest = new Sdl.SDL_Rect(x, y, anchoPantalla, altoPantalla);
    Sdl.SDL_BlitSurface(textoComoImagen, ref origen, pantalla, ref dest);
}
```

Al final del constructor de Hardware (donde inicializamos el modo gráfico) también debemos inicializar la gestión de fuentes TTF con la instrucción:

```
SdlTtf.TTF_Init();
```

Probar las fuentes

Finalmente, en el programa principal, cargamos la fuente que queramos (al principio), y en el bucle redibujamos en cada iteración el texto que queramos en la posición y color que queramos. En este ejemplo, lo mostramos en X = 200, Y = 200, color rojo:

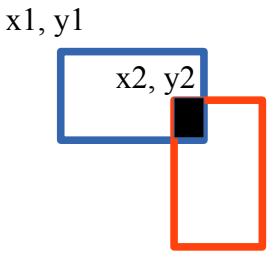
```
static void Main(string[] args)
{
   ...
   Fuente letra = new Fuente("Pacifico.ttf", 20);

    do
    {
        ...
        h.BorrarPantalla();
        h.DibujarImagen(img);
        h.EscribirTexto("Texto de prueba", 100, 300, 255, 0, 0, letra);
        ...
    } while (!terminado);
}
```

7. Detectar colisiones

En la inmensa mayoría de juegos hay que saber detectar cuándo un objeto "choca" con otro. Por ejemplo, que un proyectil impacte en una nave, o que el personaje recoja un objeto, o choque con una pared. Vamos a ver en este paso cómo podemos detectar que dos objetos colisionan entre sí.

Colisiones simples: cuadrados o rectángulos

La detección de colisiones puede ser todo lo compleja o sencilla que queramos. Por ejemplo, si estamos haciendo un comecocos estilo Pacman, podemos considerar que tanto Pacman, como los fantasmas que haya, como las paredes del laberinto, como los cocos a comer, son cuadrados o rectángulos de mayor o menor anchura y altura.



Por ejemplo, si tenemos el sprite de Pacman en la posición (*x1*, *y1*), con la anchura *ancho1* y la altura *alto1*, y el sprite de un fantasma en la posición (*x2*, *y2*), con anchura *ancho2* y altura *alto2*, Pacman colisionará con el fantasma (o al revés) si existe una zona común entre ambos rectángulos (pintada en negro en la figura derecha). Esto puede hallarse siguiendo esta estrategia: si el rectángulo rojo está más a la derecha que el azul, su coordenada x2 debe estar entre x1 y x1 + ancho1. Si está más a la izquierda, esta coordenada deberá estar entre x1 - ancho2 y x1. Lo mismo podemos decir de las coordenadas Y, según si el rectángulo rojo está por encima o por debajo del azul.

Para ayudarnos a eso, podríamos añadir un método *ColisionaCon* a nuestra clase *Imagen*, que reciba como parámetro otra imagen, y compruebe si hay colisión entre ambas, usando la estrategia anterior, que podríamos incluso simplificar en 4 comprobaciones:

```
public bool ColisionaCon(Imagen img)
{
    return (x + ancho >= img.x && x <= img.x + img.ancho &&
            y + alto >= img.y && y <= img.y + img.alto);
}
```

Prueba de las colisiones

Vamos a añadir un segundo sprite a nuestro juego, con un fantasma:

En el *Main*, lo añadiremos en una posición determinada, en concreto 100 píxeles delante de Pacman, para que lo alcance moviéndose a la derecha:

```
...
Imagen imagen = new Imagen("pacman_sprites.bmp", 50, 50);
Imagen imagen2 = new Imagen("fantasma.bmp", 50, 50);
imagen2.MoverA(200, 200);
...
```

Aprovechamos también lo visto en el apartado anterior para, si hay colisión entre los dos personajes (lo comprobamos en el *do..while* principal del juego), muestre un texto indicando que ha habido colisión:

```
do
{
    ...
    h.BorrarPantalla();
    h.DibujarImagen(imagen, coordenadas[i,0], coordenadas[i,1], 50, 50);
    h.DibujarImagen(imagen2, 0, 0, 50, 50);
    if (imagen.ColisionaCon(imagen2))
        h.EscribirTexto("Colision!!", 100, 300, 255, 0, 0, letra);
    h.VisualizarPantalla();
    Thread.Sleep(20);
} while (!terminado);
```

Otras colisiones más complejas

Existen otros tipos de juegos donde la detección de colisiones es más compleja. Si por ejemplo estamos haciendo un juego de lucha uno contra uno (tipo *Street Fighter* o similar), entonces debemos comprobar si, al hacer un golpe, la extremidad con la que lo estamos haciendo impacta en el adversario. Para eso, una forma de comprobar la colisión es marcar, en cada sprite del golpe, un cuadrado o rectángulo con la parte de ese sprite que está realizando el golpe:

Después, tendríamos que ver si ese cuadrado o rectángulo colisiona con alguna parte del sprite del adversario, aplicando un algoritmo parecido al del caso sencillo explicado al principio de este apartado.

Usos de las colisiones

El hecho de comprobar que dos elementos colisionan en un videojuego puede utilizarse para diversos propósitos:

- Si es un choque entre adversarios (pacman y su fantasma, o un luchador y su enemigo), o bien sirve para perder una vida (caso de Pacman) o para restar vida al adversario (caso de Street Fighter)
- Si es un choque con una pared, la colisión sirve para evitar que el personaje pueda moverse más allá de esa pared
- Si es un choque con un objeto, puede servir para recoger ese objeto automáticamente, y quitarlo así del escenario.
- ... etc.

8. Añadir audio

Como característica adicional que nos puede venir bien para muchos videojuegos, está la posibilidad de añadir tanto efectos de audio (por ejemplo, disparos, golpes, etc.) como música de fondo a nuestros videojuegos. Para trabajar con audios, deberemos incluir en nuestro proyecto la librería *SDL_mixer.dll*, tal y como se indicó en el apartado 1 de este tutorial.

Creación de la clase Audio

Igual que hemos hecho con otros aspectos anteriores (imágenes y fuentes, por ejemplo), vamos a implementar nuestra propia clase *Audio* que encapsule las llamadas a los métodos de Tao.Sdl para poderlos utilizar de forma más sencilla.

```
using System;
using Tao.Sdl;

namespace PruebaSDL
{
    class Audio
    {
    }
}
```

Como atributos, vamos a definir una lista donde iremos añadiendo los distintos audios que queramos reproducir (que serán de tipo *IntPtr*, al igual que las fuentes o las imágenes), y el número de canales diferentes que queramos tener, que deberá coincidir con cuántos sonidos distintos querremos reproducir a la vez en un momento del juego (normalmente 2, o como mucho 3).

```
List<IntPtr> audios;
int canales;
```

En el constructor, iniciamos el número de canales y la lista de audios, y llamamos al método *SdlMixer.Mix_OpenAudio* para indicar las características del audio a reproducir: frecuencia (normalmente 22050 o 44100), formato de audio (normalmente por defecto), número de canales y número de bytes por muestra de audio (normalmente 4096). Dejamos algunos datos configurables por parámetros.

```
public Audio(int frecuencia, , int canales, int bytesPorMuestra)
{
    this.canales = canales;
    SdlMixer.Mix_OpenAudio(frecuencia, (short)SdlMixer.MIX_DEFAULT_FORMAT, canales,
                           bytesPorMuestra);
    audios = new List<IntPtr>();
}
```

Escribimos a continuación un método que nos permita añadir un audio a la lista de audios a reproducir, a partir del nombre de archivo. Para empezar, y por simplificar, podemos usar formatos WAV y el método *SdlMixer.Mix_LoadWAV*. Haremos que el método devuelva *true* si se ha podido añadir el audio, o *false* si no se ha podido crear el objeto.

```
public bool AnyadeAudioWAV(string nombreFichero)
{
    IntPtr archivo = SdlMixer.Mix_LoadWAV(nombreFichero);
    if (archivo == IntPtr.Zero)
        return false;
```

```
    audios.Add(archivo);
    return true;
}
```

Después, añadimos un método para reproducir el audio que indiquemos (a partir de su posición en la lista, empezando por 0), por el canal que indiquemos (entre 1 y el número total de canales), y el número de reproducciones consecutivas que digamos (-1 para reproducción en bucle, útil para músicas de fondo, 0 para 1 repeticiones, 1 para 2 repeticiones, etc.).

```
public void PlayWAV(int pos, int canal, int repeticiones)
{
    if (pos >= 0 && pos < audios.Count && canal >= 1 && canal <= canales)
        SdlMixer.Mix_PlayChannel(canal, audios[pos], repeticiones);
}
```

También podemos admitir otros tipos de audio que no sean necesariamente WAV, como por ejemplo MP3 u OGG (y también WAV), usando otros métodos distintos a los anteriores. La diferencia es que no podemos elegir en qué canal reproducirlos, porque están pensados para reproducir música de fondo, y utilizarán los canales que necesiten de los que haya disponibles.

Para permitir esta versatilidad adicional en los audios, y trabajar con archivos de audio en diversos formatos, debemos utilizar el método *SdlMixer.Mix_LoadMUS* (para cargar el audio) y el método *SdlMixer.Mix_PlayMusic* (para reproducirlo). Podemos hacernos otros dos métodos alternativos para trabajar con estos, y usar los anteriores para sonidos en formato WAV monocanal, y estos nuevos para música en diversos formatos o multicanal:

```
public bool AnyadeMusica(string nombreFichero)
{
    IntPtr archivo = SdlMixer.Mix_LoadMUS(nombreFichero);
    if (archivo == IntPtr.Zero)
        return false;
    audios.Add(archivo);
    return true;
}

public void PlayMusica(int pos, int repeticiones)
{
    if (pos >= 0 && pos < audios.Count)
        SdlMixer.Mix_PlayMusic(audios[pos], repeticiones);
}
```

Probar los audios

Para probar audios en nuestro juego, añadimos un par de archivos WAV a nuestro proyecto (de la misma forma que añadimos imágenes, o fuentes TTF, haciendo que se copien siempre a la carpeta de salida), y en el programa principal creamos el objeto de tipo *Audio* y añadimos los audios a reproducir antes del bucle principal:

```
...
Audio audio = new Audio(2, 4096);
audio.AnyadeMusica("musica.wav");
audio.AnyadeAudioWAV("efecto.wav");

do
{
    ...
```

Después, la música de fondo la podemos reproducir, por ejemplo, también antes del bucle, por el canal 1, indefinidamente, y el efecto de sonido, por ejemplo, cuando pacman colisione con el fantasma, en el canal 2, y sólo una vez:

```
...
audio.PlayMusica(0, -1);

do
{
    ...
    if (imagen.ColisionaCon(imagen2))
    {
        h.EscribirTexto("Colision!!", 100, 300, 255, 0, 0, letra);
        audio.PlayWAV(1, 2, 0);
    }
    ...
} while (!terminado);
```

El código anterior tiene el problema de que, mientras persista la colisión entre pacman y el fantasma, se iniciará la reproducción del audio del efecto. Lo ideal sería cambiar a uno de los dos de posición en cuanto se detecte la colisión, o utilizar alguna variable booleana para asegurarnos de que el audio se reproduce sólo una vez.

9. Otros aspectos

En este apartado veremos algunos aspectos adicionales a considerar, que quizá no sean tan relevantes o frecuentes a la hora de desarrollar videojuegos, pero sí nos pueden venir bien cuando el videojuego es más complejo, o cuando el escenario por el que debemos movernos no cabe en la pantalla. Veremos, entre otras cosas, cómo redistribuir el código del programa principal, o cómo hacer escenarios más grandes que el propio tamaño de la pantalla, y hacer que la zona que se ve del escenario cambie según la posición del personaje. También veremos en esta sección una funcionalidad que podemos utilizar en nuestros juegos para partes del mismo: cómo dibujar diversas figuras geométricas con Tao.Sdl.

Reprogramando el bucle principal

El *do..while* del programa principal hasta ahora ha sido muy sencillo en los ejemplos vistos, pero a medida que se complica el juego se va llenando de código, hasta llegar a ser difícilmente mantenible o legible.

En ese momento, es recomendable dividir el código en métodos o funciones, de forma que cada una se encargue de una parte del trabajo a realizar en el juego. Así, un típico bucle *do..while* para un videojuego sencillo podría ser el siguiente:

```
Inicializar();
do
{
   ComprobarTeclas();
   MoverElementos();
   ComprobarColisiones();
   DibujarElementos();
   Thread.Sleep(20);
} while (!PartidaTerminada());
```

Y entonces implementaríamos en cada una de las funciones anteriores su parte del código correspondiente. Por ejemplo, en *Inicializar* crearíamos un objeto de la clase *Hardware*, cargaríamos las imágenes, etc. En *ComprobarTeclas* recogeríamos qué tecla ha pulsado el usuario, completando su acción asociada (por ejemplo, cambiar las coordenadas X e Y del personaje). Y así sucesivamente con cada función.

Hacer scroll en mapas grandes

Cuando el mapa del videojuego no cabe en el tamaño asignado para la pantalla, entonces no queda más remedio que, o bien reescalar todo el juego para que quepa (no es lo habitual, ya que se suele asignar ya una resolución de entrada apropiada), o mostrar sólo una parte de ese mapa, e irla moviendo a medida que nos movemos por él (lo habitual en juegos donde el mapa es largo, como por ejemplo en Super Mario). Esta acción de mover el mapa según nos vamos moviendo por él se denomina *scroll*, y puede ser horizontal (por ejemplo, el propio juego de Super Mario), vertical (algunos juegos de fútbol vistos desde arriba tienen ese tipo de scroll, o juegos de naves), o general (tanto horizontal como vertical, para mapas que se salen de la pantalla por todas partes, típico en juegos de rol, por ejemplo).

La filosofía a seguir cuando tenemos estos tipos de mapas es relativamente sencilla, aunque el código para implementarla pueda resultar complejo: si el personaje avanza en

una dirección, y queda mapa por mostrar en esa dirección, movemos el mapa hacia esa dirección (es decir, cambiamos el rectángulo recortado de la franja de mapa que queramos mostrar). Si ya no queda mapa por mostrar en esa dirección, seguimos moviendo el personaje, en esa dirección (sin rebasar los límites de la pantalla).

Vamos a poner un ejemplo sencillo con un scroll horizontal. Supongamos que, dado nuestro juego de tamaño 800 x 600, tenemos un mapa de tamaño 1200 x 600. Obviamente, la altura es la misma y no será necesario hacer scroll vertical, pero sí tendremos que hacer un scroll horizontal para movernos de un extremo a otro del mapa. Partiremos, por ejemplo, de uno de los laterales del mapa (por ejemplo, el borde izquierdo), y permitiremos que, según movamos el personaje a izquierda o derecha, el mapa también cambie la coordenada X del rectángulo a dibujar, a izquierda o derecha.

El programa principal podría quedar más o menos así (quitando todo lo visto anteriormente de sprites, audios, fuentes, etc. en este ejemplo, para dejar el código limitado a lo que se está explicando ahora):

```
bool terminado = false;
Hardware h = new Hardware(800, 600, 24, false);
// Fondo (de 1200 x 600, recortamos la parte visible de 800 x 600) y personaje (50x50)
Imagen mapa = new Imagen("mapa.png", 800, 600);
Imagen imagen = new Imagen("personaje.bmp", 50, 50);
// Coordenada X del personaje y del mapa (la Y la dejamos fija para simplificar)
short x = 100, xmapa = 0;

do
{
    if (h.TeclaPulsada(Hardware.TECLA_DER))
    {
        if (xmapa < 400 && x == 400)
            xmapa += 2;
        else if (x < 750)
            x += 2;
    }
    else if (h.TeclaPulsada(Hardware.TECLA_IZQ))
    {
        if (xmapa > 0 && x == 400)
            xmapa -= 2;
        else if (x > 0)
            x -= 2;
    }
    if (h.TeclaPulsada(Hardware.TECLA_ESC))
        terminado = true;
    imagen.MoverA(x, 200);

    h.BorrarPantalla();
    h.DibujarImagen(mapa, xmapa, 0, 800, 600);
    h.DibujarImagen(imagen, 0, 0, 50, 50);
    h.VisualizarPantalla();
    Thread.Sleep(20);
} while (!terminado);
```

Dentro del bucle principal, distinguimos la tecla pulsada:

- Si pulsamos la flecha derecha, moveremos el mapa siempre que quede mapa a la derecha, y el personaje esté aproximadamente en el centro de la pantalla. Si no, moveremos al personaje hasta que toque el extremo derecho, es decir, hasta que su X sea igual a la anchura de la pantalla (800) menos su propia anchura (50), es decir, 750.

- Si pulsamos la flecha izquierda, la mecánica es la misma pero hacia la izquierda (movemos el mapa a la izquierda si queda mapa en esa dirección y el personaje está en el centro del escenario, y si no, movemos el personaje hasta el borde izquierdo).
- Notar que en el método *DibujarImagen* para el mapa, recortamos el rectángulo que va desde la variable *xmapa* hasta 800 píxeles a su derecha.
- Notar también que el método *imagen.MoverA(x, 200)* que hay tras las comprobaciones de teclas sólo tendrá efecto cuando movamos al personaje, si no, lo estaremos colocando en el mismo lugar donde está. Esto se podría controlar con alguna condición, para no intentar mover al personaje innecesariamente, o llamar a este método sólo cuando variemos la variable *x*.

Otros tipos de scroll

Igual que hemos aplicado este scroll horizontal, podríamos aplicar uno vertical (simplemente dejando fijas las X y variando las Y), o uno general (variando tanto X como Y, según la tecla de dirección que se pulse en cada momento).

Scrolls continuos

Algunos tipos de juegos tienen una especie de escenarios "infinitos", que simplemente son repeticiones continuas de un mismo mapa. Para ello, lo que se hace es, una vez alcanzado el final del mapa, se dibuja por una parte un rectángulo con la parte final del mapa (empezando desde el borde izquierdo de la pantalla), y cuando acabe, desde ese mismo punto, se sigue dibujando un rectángulo con el principio del mapa, hasta llenar la pantalla. Si por ejemplo el mapa completo es este:

Al llegar al final, podemos tomar un fragmento final del mapa, y enlazarlo con un fragmento del inicio del mapa, quedando algo así:

A medida que vayamos avanzando, el primer fragmento (final del mapa) se irá acortando y el segundo fragmento (comienzo del mapa) se irá alargando, hasta volver de nuevo al inicio por completo.

Es deseable, en estos casos, que el final del mapa se pueda "enganchar" con el principio sin dar ninguna sensación de corte brusco o de salto.

Dibujar con Tao.Sdl

Aunque hemos visto cómo definir imágenes en un videojuego con Tao.Sdl, el hecho de poder dibujar diversas formas geométricas (líneas, rectángulos, círculos, etc.) con diversos colores nos puede venir bien para sustituir algunas de las imágenes que podríamos necesitar añadir. Por ejemplo, las paredes de un laberinto podríamos hacerlas con ladrillos basados en una imagen BMP o PNG, o bien dibujar rectángulos que simulen los muros del mismo.

Para poder realizar dibujos con Tao.Sdl, necesitamos añadir a nuestro proyecto la librería *SDL_gfx.dll* de la carpeta *lib* de instalación de Tao, de la misma forma que hemos añadido el resto de librerías en este tutorial.

Al añadir esta librería, dispondremos de varios métodos para dibujar diversas figuras geométricas (círculos, rectángulos, elipses, etc.), con diversos colores, en la clase *SdlGfx*. Por ejemplo, el método *rectangleRGBA* permite dibujar un rectángulo en un objeto *IntPtr* (típicamente la pantalla), dadas su coordenada X e Y (extremo superior izquierdo), su ancho, su alto (todos de tipo *short*), y su color en formato RGBA (componentes rojo, verde, azul y nivel de transparencia, todos de tipo byte). El siguiente código dibuja un rectángulo rojo en X = 200, Y = 100, de ancho 300 y alto 150, con color rojo y totalmente opaco.

```
// Orden de los parámetros: pantalla, X, Y, ancho, alto, rojo, verde, azul, transparencia
SdlGfx.rectangleRGBA(pantalla, 200, 100, 300, 150, 255, 0, 0, 255);
```

Existen muchos otros métodos disponibles, que podemos consultar desde la ayuda contextual que obtenemos al escribir el nombre de la clase *SdlGfx* en el programa. Por ejemplo, el método *filledPolygonRGBA* permite dibujar un polígono relleno de cualquier número de lados, definiendo un array con las coordenadas X e Y de cada vértice (en el orden secuencial en que se van a recorrer), el número de lados totales, y el color en formato RGBA, como en el caso anterior. El siguiente ejemplo dibuja un rectángulo que va pasando por los vértices (100, 100), (300, 100), (300, 200) y (100, 200), respectivamente, con color verde y semitransparente (120).

```
short[] vx = {100, 300, 300, 100};
short[] vy = {100, 100, 200, 200};
// Orden de parámetros: pantalla, vector de X, vector de Y, nº de puntos y color RGBA
SdlGfx.filledPolygonRGBA(pantalla, vx, vy, vx.Length, 0, 255, 0, 120);
```

Eventos en Tao.Sdl

Como hemos visto con anterioridad en el apartado 4, podemos responder a pulsaciones de teclas. En los ejemplos de código de ese apartado se utilizaban para ello métodos como *Sdl.SDL_PollEvent* y *Sdl.SDL_GetKeyState*.

Este método de control es cómodo y eficaz, pero pongámonos en otra situación. Supongamos que estamos programando un juego de lucha tipo “Street Fighter”; si queremos que cada vez que pulsemos la tecla ‘s’, nuestro personaje efectúe un

puñetazo leve, el mecanismo anterior se queda ligeramente corto, puesto que *GetKeyState* lee todos los estados de las teclas y los guarda en un array de bytes, según si están pulsadas o no. De esta forma el primer puñetazo lo haría bien, pero si dejamos pulsada la tecla 's' nuestro personaje va a estar pegando puñetazos a velocidades sobrehumanas.

Podríamos utilizar una variable booleana para controlar cúando estamos dando un puñetazo, para ignorar el resto de pulsaciones de la tecla mientras tanto, pero... ¿qué pasa cuando queremos hacer que nuestro personaje tenga combos (secuencia de teclas para realizar un ataque especial)? En este caso, los *keystates* no van a ser una alternativa viable, ya que, depende de cómo pulsemos una tecla, puede que se queden registradas 2 o más pulsaciones seguidas de la misma, y se anule así el combo que estamos haciendo.

Veremos en esta sección una alternativa para controlar tanto pulsaciones de teclas como movimientos del ratón y otros dispositivos (incluso gamepads), lo que comúnmente se conoce como **eventos**. Vamos a decirle al ordenador que esté atento a cualquier tipo de evento que pueda suceder, y si no sucede, seguirá con su transcurso normal. Así, vamos a considerar que un evento es cualquier suceso periférico de entrada que suceda, tal como bajar una tecla, subir una tecla, hacer click con un botón, soltar ese click, mover la rueda del ratón, mover el ratón en sí, etc.

Para ello utilizaremos algo que ya estábamos poniendo en nuestro código pero sin darle utilidad, que es la clase *SDL_Event*. En su versión más simple, podemos controlar si ha sucedido cualquier evento con algo como esto:

```
Sdl.SDL_Event suceso;
if (Sdl.SDL_PollEvent(out suceso) == 1)
    Console.WriteLine("Ha sucedido un evento");
```

Si probamos este código (y definimos nuestra aplicación como de tipo consola, para poder ver los mensajes de consola), veremos que cada vez que movemos el ratón, pulsamos una tecla, la soltamos, etc., detecta eventos. Y otro dato importante, si dejamos una tecla pulsada, solo detecta 1, por lo que los combos son más factibles de conseguir con esta técnica

Ya hemos conseguido tener un juego capaz de detectar eventos. Ahora, vamos a identificar tipos de eventos concretos, y si suceden, será cuando el juego haga algo concreto. Para ello, usaremos la propiedad *type* del evento, y distinguiremos el tipo de evento en concreto (tecla pulsada, tecla soltada, clic del ratón, etc.). Aquí vemos algunos ejemplos de tipos de eventos:

```
Sdl.SDL_Event suceso;

while (Sdl.SDL_PollEvent(out suceso) == 1)
{
    //Evento de salida
    if (suceso.type == Sdl.SDL_QUIT)
        Sdl.SDL_Quit();

    //Teclas pulsadas
    else if (suceso.type == Sdl.SDL_KEYDOWN)
    {
        Console.Write("Tecla pulsada ");
        switch (suceso.key.keysym.sym)
        {
            case Sdl.SDLK_s:
                Console.WriteLine("s"); break;
```

```
            default: break;
        }
    }

    //Teclas levantadas
    else if (suceso.type == Sdl.SDL_KEYUP)
    {
        Console.Write("Tecla levantada ");
        ...
    }

    //Mover el ratón
    else if (suceso.type == Sdl.SDL_MOUSEMOTION)
        Console.WriteLine(" de movimiento de ratón en coordendas"
                          + " x:{0} y:{1}", suceso.motion.x, suceso.motion.y);

    //Bajar botón del ratón
    else if (suceso.type == Sdl.SDL_MOUSEBUTTONDOWN)
    {
        Console.Write("Click de raton");
        switch (suceso.button.button)
        {
            case Sdl.SDL_BUTTON_LEFT:
                Console.WriteLine(" en el boton izquierdo"); break;
            default: break;
            //Puede ser también Sdl.SDL_BUTTON_RIGHT
        }
    }
}
```

Hay bastantes cosas nuevas, pero muy intuitivas. Por ejemplo, el evento *Sdl.SDL_QUIT* sirve para detectar un intento de cerrar la aplicación. En versiones de la librería para C/C++ recogía cualquier intento de que el jugador quiera cerrar el juego, como pulsar la "x" de la ventana, o pulsar alt+f4. Pero en C# parece que sólo funciona adecuadamente al pulsar la X, que por otra parte, es un evento importante.

El resto de eventos hablan por si solos: *KEYDOWN* es pulsar una tecla, *KEYUP* levantarla o soltarla, *MOUSEMOTION* recoge cualquier movimiento del ratón, *MOUSEBUTTONDOWN* recoge cualquier pulsación de cualquier botón del ratón y *MOUSEBUTTONUP* (que no se ha contemplado en el código de ejemplo) recogería el evento de dejar de pulsar el botón de ratón previamente pulsado. Además, existen muchos mas tipos de eventos como el scrolling de rueda o joysticks, y se puede profundizar un poco más en ellos con la ayuda contextual que aparece a medida que utilizamos las clases y métodos de SDL.