

SDL con cafeína

Metiendo mano a la POO

Belén Albeza (BenKo)

9 de junio de 2006

Licencia

Esta obra está bajo una licencia Reconocimiento - NoComercial - CompartirIgual 2.5 Spain de **Creative Commons**. Para ver una copia de esta licencia, visita la web <http://creativecommons.org/licenses/by-nc-sa/2.5/es/> o envía una carta a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

1. ¿Por qué POO y no SDL a pelo?

Si eres de la vieja escuela de C, la simple mención de C++ probablemente te haga rechinar los dientes y te cause espasmos incontrolados¹ De hecho, las librerías SDL están programadas en un principio para C, así que no es obligatorio usar C++. Pero...

La **Programación Orientada a Objetos** (POO) nos ofrece un abanico bestial de ventajas a la hora de desarrollar videojuegos: mejor reutilización de código, encapsulamiento, mayor claridad, etc. Eso, y la **Standard Template Library** que nos proporciona muchos TAD's populares: listas, pilas, colas, tablas hash, vectores dinámicos, la clase string², etc.

En este capítulo de **SDL con cafeína** transformaremos el programa del capítulo previo (cargaba una imagen y la mostraba en pantalla) para que pase a usar clases y objetos. Además, aprenderemos a limitar los FPS y a emplear la técnica del *double buffering*.

¡Así que manos a la obra!

¹Como dice el proverbio, "*C++ es a C lo que el cáncer de pulmones es a los pulmones*".

²Porque, reconozcámoslo, ya estamos en el siglo XXI y no es sano estar reservando y liberando memoria a mano para utilizar una simple cadena de caracteres.

2. Empezando el Manager

2.1. ¿Qué demonios es eso?

En el motor de un videojuego necesitamos “algo” que se llama *resources manager* y que nos hace la vida más fácil. Se encarga de controlar todos los **recursos** (gráficos, músicas, etc), liberarlos cuando ya no hagan falta, etc. Además, nuestro *manager* también se encargará de operaciones básicas como inicializar el modo de vídeo o controlar la visualización de *frames*.

2.2. El patrón singleton

Vamos a crearnos una clase llamada **Manager**, y como de costumbre, separaremos su código en dos ficheros: `manager.h` y `manager.cc`. La característica más singular de esta clase es que sigue el patrón de diseño **singleton**, es decir, que sólo habrá una única instancia de esta clase en ejecución. Esto se consigue con una triquiñuela muy simple utilizando métodos y variables estáticos.

Una versión preliminar de la definición de la clase sería esta:

```
class Manager {
public:
    static Manager* GetManager();
    ~Manager();
private:
    Manager();
    static Manager* instance;
};
```

Como ves, el truco está en poner el **constructor** en la **parte privada**, para asegurarnos de que el usuario no pueda crear directamente instancias de la clase. La instancia de **Manager** se crea/obtiene con la función **estática** `GetManager()`.

La implementación de esto sería:

```
//inicializamos el puntero a NULL
Manager* Manager::instance = NULL;

//devuelve un puntero a la única instancia del Manager
//que está en ejecución. Si no hay ninguna, la crea
Manager* Manager::GetManager() {
    if(instance == NULL) {
        instance = new Manager();
    }
}
```

```
    return(instance);  
}
```

Así, si queremos llamar a un método de la clase, pondríamos (sin preocuparnos de crear una instancia antes, ya que si no existiese se crearía automáticamente):

```
Manager::getManager()->NombreDelMetodo();
```

2.3. Inicialización del modo de vídeo (reloaded)

Para inicializar el modo de vídeo, nos crearemos un método llamado `SetMode`, al que le pasaremos la resolución de pantalla y si lo queremos *fullscreen* o en ventana. Vamos a hacer que la profundidad de paleta sea siempre 16 bits. Si no te gusta, siempre puedes añadir un parámetro más a `SetMode`. Además de esto, `SetMode` se encargará de inicializar las SDL él solito.

Nos harán falta tres **atributos privados** para la clase `Manager`:

- `int screen_w`: resolución horizontal
- `int screen_h`: resolución vertical
- `SDL_Surface *screen`: puntero a la pantalla

La implementación del método sería así (pégale un repaso al capítulo anterior si hay algo que no recuerdas):

```
//inicializa el modo de vídeo  
//devuelve TRUE si hubo éxito  
//entrada: ancho, alto y fullscreen  
bool Manager::SetMode(int width, int height, bool full) {  
  
    bool res = false;  
    if(SDL_Init(SDL_INIT_VIDEO) == 0) { //inicializar SDL  
        atexit(SDL_Quit);  
  
        if(full) { //pantalla completa  
            screen = SDL_SetVideoMode(width,height,16,  
                SDL_SWSURFACE | SDL_FULLSCREEN);  
        } else { //ventana  
            screen = SDL_SetVideoMode(width,height,16,  
                SDL_SWSURFACE);  
        }  
        if(screen!=NULL) {  
            res = true;  
        }  
    }  
}
```

```

        screen_w = width;
        screen_h = height;
    }
}
return(res);
}

```

El método devuelve `true` si todo ha ido bien y `false` si se produjo algún error. Cosillas a tener en cuenta:

Al inicializar las SDL con `SDL_Init`, sólo le pasamos el flag de vídeo. Conforme vayamos incorporando cosas a nuestro motor (como por ejemplo, sonido), tendremos que ir añadiendo flags.

También fíjate en que al llamar a `SDL_SetVideoMode` le indicamos que use una *surface* software. Ya hablamos de esto en el anterior capítulo: haz pruebas con *surfaces* hardware para ver si te convienen más o no.

Una vez que desde nuestro programa hayamos llamado a este método, la instancia de `Manager` tendrá actualizados los atributos de `screen_w` y `screen_h`, además de que `screen` contendrá un puntero a pantalla.

Por cierto, no vendría nada mal crear un par de métodos *get* para obtener los valores de `screen_w` y `screen_h`, además de otro llamado `Screen` que devuelva un puntero a la pantalla³.

2.4. El título de la ventana

Un método bastante majo que podemos implementar es `SetTitle`, que se encarga de poner un título a la ventana en la que se ejecute el juego (no lo veremos si estamos a pantalla completa, obviamente). Es simplemente un *wrapper* de la función SDL llamada `SDL_WM_SetCaption`:

```

//pone un título a la ventana
void Manager::SetTitle(string title) {
    SDL_WM_SetCaption(title.c_str(), NULL);
}

```

Una pregunta muy legítima podría ser esta: “¿Para qué #@%\$ gasto tiempo en re-implementar una función para simplemente cambiarle el nombre!?”. No es una pérdida de tiempo, piénsalo bien. Quizás un día estés aburrido y decidas cambiarte a Allegro o reimplementar el motor usando OpenGL. ¿Qué sería de tus juegos entonces? Es mejor que desde el código del juego **no hagas ninguna llamada ni uses ninguna cosa**

³A no ser, claro, que seas alérgico a poner todos los atributos en la parte privada de la clase. Los talibanes de la POO seguramente te llamen hereje, pero eh, hay cosas peores.

relacionada con SDL. Así, si cambias la reimplementación interna del motor, tu juego ni se enterará⁴.

3. Control de FPS

En un videojuego es extremadamente importante controlar a qué **velocidad** se ejecuta. Esta velocidad se mide en **imágenes por segundo**, unidad más conocida como FPS (*frames per second*). Cuantos más FPS, más fluido se verá el juego y menos cansará la vista. Se ha determinado que la cantidad de FPS mínimos que necesita el ojo humano para ver una animación “sin saltos” es de 24 FPS, aunque en los videojuegos es deseable alcanzar cifras superiores.

Lógicamente, la cantidad de FPS que podemos mostrar viene determinada por la capacidad del hardware. Si no controlamos los FPS, los videojuegos irán muy rápidos en ordenadores potentes, y muy lentos en ordenadores antiguos. No queremos esto, ya que puede causar que el videojuego sea injugable.

3.1. Timers

SDL dispone de una función que devuelve el nº de milisegundos transcurridos desde que se inició el programa: `SDL_GetTicks()`. Pero como ya hemos comentado que tenemos que huir de utilizar directamente funciones propias de SDL, vamos a crearnos una clase para controlar el paso del tiempo (y la podremos utilizar tanto en el motor como en el juego en sí).

Esta clase se llama `Timer` (temporizador), y funcionará a modo de cronómetro: tendrá un método para iniciar el timer y otro para obtener el nº de ticks (milisegundos) que han transcurrido desde que le “dimos al start”. La definición de la clase sería:

```
class Timer {
public:
    //forma canónica de la clase
    Timer();
    Timer(const Timer&);
    ~Timer();
    Timer operator=(const Timer&);

    void Start();
    unsigned int GetTicks();
private:
    unsigned int start_ticks;
```

⁴Evidentemente esto es en el mundo donde caminas entre nubes de algodón de azúcar. En la Vida Real, seguramente te toque cambiar parte del código del juego. Pero la gracia está en que no tendrás que cambiar *todo* el juego.

```
};
```

Como ves, se trata de una clase muy sencilla y no haría falta hacer el constructor de copia, pero siempre es una buena costumbre programar a mano la **forma canónica** de las clases (constructor por defecto, constructor de copia, destructor y la sobrecarga de la asignación).

El método `Start` asigna al atributo `start_ticks` (que se inicializa a 0 en el constructor) el valor que devuelve `SDL_GetTicks`. Más adelante, podremos obtener el nº de ticks que han transcurrido desde que invocamos a `Start` simplemente restando los ticks actuales menos los iniciales.

```
//inicia el timer
void Timer::Start() {
    start_ticks = SDL_GetTicks();
}

//devuelve los ticks que han pasado
unsigned int Timer::GetTicks() {
    int res = 0;
    if(start_ticks > 0) {
        res = SDL_GetTicks() - start_ticks;
    }
    return(res);
}
```

3.2. Limitación de los FPS

En cada videojuego que hagamos tendremos un **bucle principal** que se encargue de recoger los eventos generados por el usuario, actualizar el mundo del juego y los personajes, y renderizar todo en pantalla. Es en este bucle donde tenemos que poner límite a los FPS. Para ello, vamos a añadir un método a la clase `Manager`, que se llame `WaitFrame` y que simplemente no hará nada hasta que toque dibujar el siguiente *frame*. Necesitaremos un método (`SetFPS`) para establecer el nº de FPS que queremos, y los siguientes atributos privados:

- `int fps`: nº de fps por segundo que queremos
- `unsigned int old_ticks`: instante de tiempo en el que se produjo el frame anterior
- `Timer fps_timer`: *timer* para controlar los FPS

El *timer* lo iniciaremos en el método `SetMode` que ya tenemos, y asignaremos el valor que tenga al inicio a la variable `old_ticks`, para inicializarla. En el constructor de `Manager` pondremos el valor por defecto que queramos al atributo `fps`.

Con todo lo anterior, el método WaitFrame nos quedaría así:

```
//espera hasta el siguiente frame
void Manager::WaitFrame() {
    while((fps_timer.GetTicks()-old_ticks)
           < ((float) 1.0/fps*1000)) {
        //no hacer nada
    }
    old_ticks=fps_timer.GetTicks(); //actualizar old_ticks
}
```

Vamos a ver cómo quedaría todo lo que hemos hecho hasta ahora. El siguiente programa mueve una imagen horizontalmente y termina cuando la imagen sale de la pantalla. Prueba al cambiar los FPS para ver la diferencia.

```
#include "timer.h"
#include "manager.h"
#include <iostream>

using namespace std;

int main(int argc, char *argv[]) {
    if(Manager::GetManager()->SetMode(640,480,false)) {
        Manager::GetManager()->SetFPS(24); //fps

        //cargar la imagen
        SDL_Surface *pic = SDL_LoadBMP("alien1.bmp");
        if(!pic) {
            cout<<"("<<endl;
            return(0);
        }

        //rectángulo destino
        //empezamos en las coordenadas 0,100
        SDL_Rect dest;
        dest.x = 0;
        dest.y = 100;
        dest.w = pic->w;
        dest.h = pic->h;

        while(dest.x<640) {
            //borra la pantalla
            SDL_FillRect(Manager::GetManager()->Screen(),
                        NULL,0x000000);

            //pinta la imagen

```

```

SDL_BlitSurface(pic, NULL,
                Manager::GetManager()->Screen(),
                &dest);
//actualizamos la pantalla
SDL_UpdateRect(Manager::GetManager()->Screen(),
               0,0,0,0);

//movemos el sprite
dest.x+=5;

//control de FPS
Manager::GetManager()->WaitFrame();
}
}
}

```

El código fuente de este ejemplo (y de lo que llevamos hasta ahora de las clases) se encuentra en el directorio `singlebuffer` de los archivos de ejemplo que acompañan el tutorial. Para compilarlo, simplemente ejecuta `make` desde una consola de comandos⁵.

Todo el código anterior debería serte familiar y fácil de comprender. La única sentencia que aún no hemos visto es:

```

SDL_FillRect(Manager::GetManager()->Screen(), NULL,
             0x000000);

```

Esa sentencia se encarga de “borrar” la pantalla en cada frame (en realidad rellena todo el rectángulo de la pantalla de color negro).

4. Double buffering

4.1. Implementación

Si has compilado y ejecutado el programa anterior⁶, habrás observado un molesto parpadeo. Esto se debe a que cuando el monitor **refresca** la imagen (aprox. a 70Hz) a nosotros nos ha pillado a mitad de nuestras operaciones de dibujo. Así que se muestran cosas del *frame* anterior, y cosas del actual. Queda muy feo y muy cutre.

Afortunadamente, existe una técnica muy fácil de utilizar que resuelve esto: el **buffer doble** (*double buffering* en inglés). Un *buffer* es como un folio en blanco, y nosotros tenemos dos. El jugador sólo ve el folio de delante (el *front buffer*), así que nosotros podemos pintarrajear lo que queramos en el folio de detrás (el *back buffer*). Una vez que

⁵Si eres de Windows y usas Dev-C++ simplemente crea un proyecto como vimos en el capítulo 0

⁶Si no lo has hecho aún, hazlo. *Ahora*.

hayamos acabado todas las operaciones de dibujo para el *frame* actual, intercambiamos los folios (operación *flipping*), así que el *back buffer* pasa a ser el folio que ve el usuario, y nosotros nos quedamos el que antes era el *front buffer* para pintar de nuevo en la sombra.

En pseudocódigo, nuestro bucle principal del juego quedaría así:

```
Repetir
  Borrar la pantalla
  Actualizar los objetos del juego
  Pintar los objetos del juego en el backbuffer
  Intercambiar buffers
  Esperar al siguiente frame
FinRepetir
```

Para poder utilizar el *double buffering* en SDL, simplemente tenemos que pasarle el *flag* `SDL_DOUBLEBUF` a la función `SDL_SetVideoMode`, además de tener que crear la pantalla como una *surface* hardware⁷. Vamos a modificar el método `SetMode` de la clase `Manager`, cambiando las líneas que inicializaban el vídeo por estas otras:

```
if(full) { //pantalla completa
    screen = SDL_SetVideoMode(width,height,16,SDL_DOUBLEBUF
        | SDL_HWSURFACE | SDL_FULLSCREEN);
} else { //ventana
    screen = SDL_SetVideoMode(width,height,16,SDL_DOUBLEBUF
        | SDL_HWSURFACE);
}
```

Con esto ya dibujaremos siempre sobre el *backbuffer*. Para intercambiar los *buffers*, simplemente llamamos a la función `SDL_Flip`, a la que le pasaremos el puntero a la pantalla. Vamos a crearnos un método público nuevo llamado `Frame`, que se encargue de intercambiar los *buffers* y esperar hasta el siguiente *frame*. Este será el método que llamemos desde nuestro juego, así que podemos meter a `WaitFrame` en la parte privada de la clase `Manager`. El código para `Frame` nos quedaría así:

```
//intercambia buffers y espera hasta el siguiente frame
void Manager::Frame() {
    SDL_Flip(screen);
    WaitFrame();
}
```

Como ves, ya no hace falta llamar a la función `SDL_UpdateRect` para actualizar la pantalla: al intercambiar los buffers se hace automáticamente.

Para terminar el capítulo, vamos a crear un método público en `Manager` que limpie la pantalla. El numerito extraño que aparece como último parámetro es el valor en hexadecimal que corresponde al color negro puro:

⁷Esto no implica que todas las demás *surfaces* deban ser de hardware también.

```

//borra en negro toda la pantalla
void Manager::ClearScreen() {
    SDL_FillRect(screen, NULL, 0x000000);
}

```

4.2. Ejemplo

Vamos a transformar el programa que hicimos para controlar los FPS para que use el *double buffering*. En realidad sólo hay que hacer estos cambios:

- Cambiar la llamada a `Manager::WaitFrame()` por una a `Manager::Frame()`.
- Eliminar la llamada a `SDL_UpdateRect`.
- Cambiar la llamada a `SDL_FillRect` por una a nuestro método `ClearScreen`, ya que estamos.

El bucle principal quedaría así:

```

while(dest.x<640) {
    //borra la pantalla
    Manager::GetManager()->ClearScreen();

    //pinta la imagen
    SDL_BlitSurface(pic, NULL,
                   Manager::GetManager()->Screen(),
                   &dest);

    //movemos el sprite
    dest.x+=5;

    //mostrar imagen y esperar al siguiente frame
    Manager::GetManager()->Frame();
}

```

Puedes encontrar este ejemplo, así como el código fuente de las clases que hemos implementado en el archivo comprimido con los ficheros de ejemplo. Viene un fichero `Makefile` para que puedas compilar con `make`.