MulticocoSDL

José Ladislao Lainez Ortega y José Molina Colmenero

15 de mayo de 2013

Índice

1. Introducción **2**

2. Bibliotecas **3**
2.1. SDL . . . 3
2.2. SDL_Mixer . . . 3
2.3. SDL_TTF . . . 4

3. Imagen **5**
3.1. Window . . . 5
3.1.1. Inicialización . . . 5
3.1.2. Creación . . . 5
3.1.3. Renderizado . . . 6
3.1.4. Eliminación . . . 7
3.2. Sprite . . . 7
3.3. SpriteSheet . . . 7

4. Audio **8**
4.1. Sound . . . 8
4.2. Music . . . 8

5. Lógica **9**
5.1. Vector2D . . . 9
5.2. CollisionBox . . . 9
5.3. Entity . . . 9

6. Juego **10**
6.1. Inicialización . . . 10
6.2. Bucle principal . . . 10
6.3. Eventos . . . 10

1. Introducción

MulticocoSDL es un juego arcade que emula al famoso Pacman realizado como proyecto para la asignatura Sistemas Multimedia del Grado de Ingeniería Informática de la Universidad de Jaén.

El objetivo de MulticocoSDL es alcanzar varias areas multimedia en un mismo programa haciendo uso de los tres elementos principales del software multimedia:

Emplazamiento espacial
: Los distintos elementos visuales (enemigos, escenario, pacman) son dibujados sobre un canvas en posiciones especficas y adems se puede mover por l.

Control temporal
: El jugador puede moverse por el escenario a una determinada velocidad as como los enemigos, estando todos ellos animados.

Interacción
: Mediante el teclado el jugador puede dar órdenes a pacman de forma que este se mueva en la dirección que el jugador le indica.

El código se encuentra alojado en GitHub en el siguiente enlace y contiene un archivo de proyecto de XCode para Mac OS X, si bien el código es portable a Windows y Linux.

`https://github.com/L4D15/MulticocoSDL`

2. Bibliotecas

Se ha hecho uso de la biblioteca Simple DirecMedia Layer así como de algunos submódulos de esta para trabajar con el renderizado de imágenes, texto y reproducción de audio. A continuación explicamos qué tareas ha realizado cada una.

Para más informacin sobre Simple DirecMedia Layer:

`http://www.libsdl.org/`

2.1. SDL

Biblioteca con las operaciones básicas para crear una ventana y dibujar en ella. Algunas de las utilidades más usadas han sido:

SDL_Rect
: Estructura para definir un rectángulo. Usado a la hora de recorta un área de un SpriteSheet y dibujar en un canvas.

SDL_Surface
: Superficie o canvas sobre el que dibujar, usado no solo en la ventana principal como contexto de renderizado, sino también para almacenar en la memoria de la tarjeta gráfica los distintos Sprites a renderizar.

SDL_BlitSurface(SDL_Surface*,SDL_Rect*,SDL_Surface*,SDL_Rect*)
: Mediante esta función se puede dibujar un área seleccionada de un canvas origen en un área de un canvas de destino. Se ha usado tanto para dibujar en la ventana principal como para separar los distintos Sprites del SpriteSheet.

SDL_LoadBMP(const char*)
: Como bien indica su nombre carga una imagen en formato BMP, la guarda en memoria gráfica y devuelve un puntero a una SDL_Surface, de modo que podamos usar la imagen cargada.

Esta biblioteca puede descargase desde el siguiente enlace:

`http://www.libsdl.org/download-1.2.php`

2.2. SDL_Mixer

Biblioteca modular de SDL que facilita el trabajo con la reproducción de sonido y música. Los elementos más destacados de esta biblioteca son:

Mix_Chunk
: Contenedor de sonido, igual que SDL_Surface lo era de imágenes.

Mix_Music
: Contenedor de sonido específico para música.

Mix_LoadWAV(const char*)
: Carga un archivo en formato WAV y devuelve un puntero a Mix_Chunk para poder trabajar con el audio del archivo.

Mix_LoadMUS(const char*)
: Carga un archivo en formato WAV, OGG, MP3 o FLAC devolviendo un puntero a Mix_Music. Esta función es específica para cargar música ya que SDL_Mixer trabaja de forma distinta los sonidos y la música.

La biblioteca se puede descargar desde:

`http://www.libsdl.org/projects/SDL_mixer/`

2.3. SDL_TTF

A la hora de mostrar la puntuación del jugador necesitábamos mostrar texto por pantalla, por lo que recurrimos a esta biblioteca (otro módulo del proyecto SDL) específica para mostrar texto por pantalla. Una curiosidad sobre esta biblioteca es que hace uso de fuentes TTF en lugar de recurrir a fuentes en archivos bitmap como sucede, por ejemplo, al renderizar texto en OpenGL.

De esta biblioteca hemos usado:

TTF_Font
: Contenedor para la información de la fuente a usar a la hora de renderizar el texto.

TTF_OpenFont(const char*, int)
: Carga una fuente desde el archivo en formato TTF especificado y usando el tamao indicado.

TTF_RenderText_Solid(TTF_Font, const char*, SDL_Color)
: Crea una superficie sobre la que dibuja el texto pasado usando la fuente indicada y el color deseado. Una vez tengamos esa superficie habrá que dibujarla usando la función SDL_BlitSurface(...).

Esta biblioteca está disponible en:

`http://www.libsdl.org/projects/SDL_ttf/`

3. Imagen

En esta sección explicaremos las clases relacionadas con el apartado visual de la aplicación, desde la creación de la ventana hasta la animación de los personajes que aparecen en pantalla.

3.1. Window

Antes de poder dibujar nada en pantalla primero necesitamos una ventana para nuestra aplicación.

3.1.1. Inicialización

SDL nos proporciona facilidades a la hora de crear una ventana preparada para renderizar imágenes, pero antes de poder usarlas necesitamos inicializar SDL.

```
int err = SDL_Init(SDL_INIT_AUDIO | SDL_INIT_VIDEO);

    if (err < 0) {
        // Mostrar error
        std::cout << "Error al inicializar SDL: " <<
           SDL_GetError() << std::endl;
        exit(1);
    }
```

Mediante los flags SDL_INIT_AUDIO y SDL_INIT_VIDEO le indicamos que nuestra aplicación va a hacer uso de estas dos funcionalidades. Si no indicaramos, por ejemplo, el flag de audio, nuestra aplicación no reproduciría audio.

Si hubiera algún problema durante la incialzación, SDL puede darnos información sobre el fallo mediante SDL_GetError(). El uso de esta función para obtener información sobre fallos durante el uso de SDL será una constante a lo largo del proyecto, por lo que se omitirá el tratamiento de errores de ahora en adelante. Para más detalles sobre ello consultar el código fuente.

3.1.2. Creación

Ahora que SDL ya está preparado para funcionar procedemos a crear nuestra ventana.

```
this->_screen = SDL_SetVideoMode(w, h, 16, SDL_HWSURFACE |
   SDL_DOUBLEBUF);
```

Donde _screen es un SDL_Surface*, w es la anchura y h la altura de nuestra ventana (y por consiguiente la resolución en pixeles de nuestro canvas), 16 es la profundidad de bits por pixel (o bpp) y los ltimos dos son dos flags que indican lo siguiente:

- SDL_HWSURFACE — La información es guardada en la memoria de la gráfica, de forma que es más rápido trabajar con ella.
- SDL_DOUBLEBUF — Indica que este canvas tiene doble buffer. Esto es importante para que podamos dibujar sobre un buffer mientras el otro se está mostrando y una vez terminamos de dibujar se intercambian los buffers. De esta forma evitamos que el usuario vea cómo se dibujan los elementos poco a poco.

3.1.3. Renderizado

Una vez la ventana ha sido creada debemos mostrar los elementos que componen el juego. Si bien el método render de la clase Windows contiene mucho más código, nuestro objetivo aquí es explicar su funcionamiento, por lo que presentamos una versión resumida en la que se renderiza un objeto de ejemplo (clase Entity explicada más adelante) llamado object.

```
void Window::render()
{
    SDL_FillRect(this->_screen, NULL, SDL_MapRGBA(this->
       _screen->format, 0, 0, 0, 255));

    object.render(this->_screen);

    SDL_Flip(this->_screen);
}
```

La primera función llena el canvas con color negro. Los parámetros que se le pasan son una SDL_Surface*, que será la superficie a rellenar con color; después un SDL_Rect* que indicará el area que queremos rellenar y que al pasarle NULL lo que hace es coger toda el area; y por último el color del que queremos rellenar el area, que en este caso será negro.

Después llamamos al método render del objeto al que le pasamos un SDL_Surface*, es decir, la superficie en la que queremos dibujarlo. Como queremos dibujarlo en la ventana, le pasamos el puntero al canvas creado anteriormente y que tenemos guardado en el atributo _screen de nuestra clase.

Por último intercambiamos los buffers de la ventana.

3.1.4. Eliminación

Hay que tener cuidado al trabajar con las SDL_Surface pues indicamos a SDL que guarde la información visual de nuestra ventana en la memoria gráfica. Esto quiere decir que si hicieramos esto:

```
delete this->_screen;
```

Estaríamos dejando basura en la memoria gráfica, ya que delete borra la información de la memoria principal, pero no sabe trabajar con la memoria gráfica. Es por esto que cuando no necesitemos más una SDL_Surface se debe liberar usando una función específica de SDL que se encarga tanto de eliminarla de memoria gráfica como de memoria principal (la información visual, es decir, los pixels, están en memoria gráfica, pero otra mucha información se guarda en memoria principal también).

Para lidiar con esto hay que declarar los destructores de las clases que hagan uso de alguna SDL_Surface para que la liberen antes de ser borrados. Mostramos aquí el de la clase Window y omitiremos esta explicación para el resto de clases.

```
Window::~Window()
{
    SDL_FreeSurface(this->_screen);
    SDL_Quit();
}
```

La función SDL_Quit() debe llamarse una vez nuestra aplicación vaya a finalizar y no necesitemos más SDL, pues liberará los recursos que internamente haya reservado cuando lo inicializamos. Como nuestra aplicación está vinculada a la existencia de la ventana, una vez esta sea destruida debemos "terminar" con SDL, por eso se ha incluido en el destructor.

3.2. Sprite

3.3. SpriteSheet

4. Audio

4.1. Sound

4.2. Music

5. Lógica

5.1. Vector2D

5.2. CollisionBox

5.3. Entity

6. Juego

6.1. Inicialización

6.2. Bucle principal

6.3. Eventos