MulticocoSDL

José Ladislao Lainez Ortega y José Molina Colmenero

18 de mayo de 2013

Índice

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. Clases

De forma resumida presentamos aquí un esquema de cómo se relacionan las clases creadas en el proyecto. Más adelante presentamos una descripción más detallada de cada clase en relación a un area multimedia.

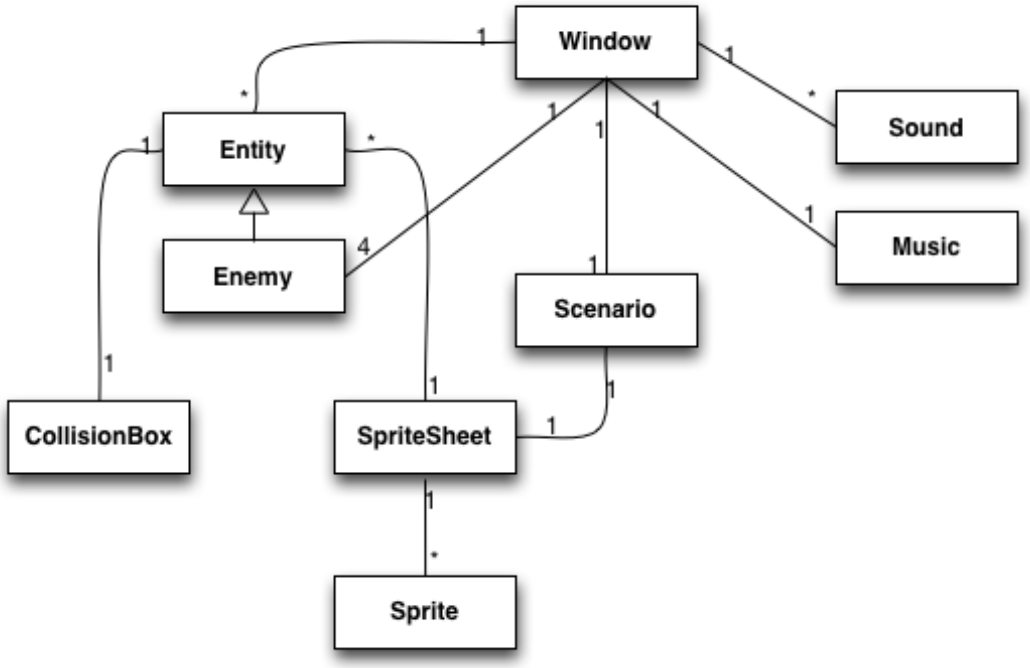

Figura 1: Clases del proyecto

4. 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.

4.1. Window

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

4.1.1. 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.

4.1.2. 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 últimos dos son 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.

4.1.3. 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.

4.1.4. 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.

4.2. Sprite

Los Sprites serán las imágenes que representen los objetos de nuestro juego. Estos Sprites podrán estar animados, para lo cuál necesitarán una imagen preparada para ello. Nuestra aplicación soportará Sprites animados siempre y cuando todos los frames tengan las mismas dimensiones.

De esta forma a la hora de renderizar el sprite se renderizará solo una sección de la imagen correspondiente al frame actual de la animación.

Figura 2: Ejemplo de Sprite

Para manejar esta sistema de animación necesitamos algo más que simplemente llamar a la función SDL_BlitSurface(...) de SDL, y ese es el cometido de la clase Sprite.

```
class Sprite
{
public:
                    Sprite(SDL_Surface* img, int animations, int w, int h);
                    ~Sprite();

    void            render(SDL_Surface* screen, Vector2D& pos);
    void            nextFrame();
    void            setFrameSkip(unsigned int f);

private:
    SDL_Surface*    _bitmap;
    unsigned int    _width;
    unsigned int    _height;

    unsigned int    _nAnimations;
    SDL_Rect*       _frames;
    unsigned int    _currentFrame;
    unsigned int    _frameSkip;
    unsigned int    _framesSkipped;
};
```

Cada uno de los atributos tiene un cometido específico:

_bitmap
: Contiene la información de la imagen guarda en memoria gráfica. Se trata de una sola imagen con todos los frames de la animación.

_width y _height
: Ancho y alto de cada frame individual que componen la animación.

_nAnimations
: Número de frames que componen la animación.

_frames
: Rectángulos que corresponden al area de la imagen de cada frame de la animación. Por ejemplo, si recortamos la imagen _bitmap usando el rectángulo de la primera posición, obtendremos el primer frame de la animación, si usamos el de la segunda posición, el segundo frame y así sucesivamente.

_currentFrame
: Indica en qué frame de la animación nos encontramos actualmente. De este modo cuando se llame al método render se renderizará el frame que corresponde a la animación actual de forma automática.

_frameSkip y _frameSkipped
: Usados para controlar la velocidad de animación. Cuanto mayor sea el frameSkip, más lenta será la animación.

4.2.1. Transparencia

Veamos cómo podemos asignar transparencia al Sprite cuendo se crea:

```
Sprite::Sprite(SDL_Surface* img, int animations, int w, int h)
{
        // Asignacion de atributos [omitido]
        // ...

        SDL_SetColorKey(
                this->_bitmap,
                SDL_SRCCOLORKEY,
                SDL_MapRGB(this->_bitmap->format,255,0,255)
        );

        // Creacion de los rectangulos de cada frame [omitido]
        //...
}
```

Nos interesa analizar el uso de SDL_SetColorKey. Esta función lo que hace es indicar a la imagen cual es su color clave. El color clave es aquel que se usa como transparencia, de modo que a la hora de dibujar la imagen se dibuja todo menos ese color. En nuestro caso y como se hace en la mayoría de los Sprites de videojuegos hemos usado como color clave un magenta puro, que corresponde con los valores R = 255, G = 0 y B = 255. Se puede ver este color como fondo en la imagen de la moneda presentada anteriormente.

4.2.2. Renderizado y animación

Respecto al renderizado y la animación debemos hablar de dos métodos de la clase Sprite: render y nextFrame.

```
void Sprite::render(SDL_Surface *screen, Vector2D& pos)
{
    SDL_Rect source = this->_frames[this->_currentFrame];
    SDL_Rect destination;
    destination.x = pos.x();
    destination.y = pos.y();
    destination.w = this->_width;
    destination.h = this->_height;

    SDL_BlitSurface(this->_bitmap, &source, screen, &destination);
}
```

Podemos ver que el rectángulo correspondiente con el frame actual (_currentFrame) se obtiene de la colección de rectángulos que creamos durante el constructor, y el rectángulo de destino que define el area donde se dibujará en el canvas de destino se calcula a partir de la posición y del ancho y alto del frame. Después solo hace falta una llamada a la ya mencionada función SDL_BlitSurface(...) y nuestro Sprite se dibujará.

¿Y cómo se anima el Sprite si el índice _currentFrame no avanza? Para eso está el método nextFrame que hace que la animación del Sprite avance siguiendo las restricciones impuestas por el frame skip indicado.

```
void Sprite::nextFrame()
{
    if (this->_framesSkipped == this->_frameSkip) {
        this->_currentFrame = (this->_currentFrame + 1) % this->_nAnimations;
        this->_framesSkipped = 0;
    } else {
        this->_framesSkipped++;
    }
}
```

4.3. SpriteSheet

Nuestra clase Sprite solo puede manejar una animación por Sprite, lo cual nos limita bastante a la hora de animar objetos más complejos que una moneda, como por ejemplo,

Figura 3: Spritesheet de Pacman

a Pacman o a los fantasmas. Para ello haremos uso de la clase SpriteSheet que gestionará varios Sprites para un mismo archivo y que además nos facilitará el manejo de las animaciones.

Nuestra clase debe manejar varias animaciones y además, con números de frames distintos.

```
SpriteSheet::SpriteSheet(const char* img,
                         int w, int h,
                         int* animations, int nAnimations)
{
    SDL_Surface* spriteSheet = SDL_LoadBMP(img);
    SDL_Surface* currentSprite;
    for (int i = 0; i < nAnimations; i++) {
        // Creamos una nueva superficie vacia
        currentSprite = SDL_CreateRGBSurface(
                                        SDL_HWSURFACE,
                                        w * animations[i], h,
                                        16, 0, 0, 0, 0);
        // Area del SpriteSheet a copiar en el Sprite
        SDL_Rect origin;
        origin.x = 0;
        origin.y = i * h;
        origin.w = w * animations[i];
        // Dibujamos parte del SpriteSheet
        SDL_BlitSurface(spriteSheet, &origin, currentSprite, NULL);

        // Insertamos un nuevo Sprite en la coleccion
        this->_sprites.push_back(new Sprite(currentSprite,animations[i],w,h));
    }
    SDL_FreeSurface(spriteSheet);
}
```

De esta forma hemos dividido la imagen en Sprites que corresponden a cada una de las filas (animaciones) del SpriteSheet.

4.3.1. Vincular animaciones

Tenemos las distintas animaciones del SpriteSheet separadas pero, ¿Cómo elegimos qué animación queremos reproducir? SpriteSheet guarda un índice con la posición en la colección donde se encuentra la animación actual, que será la que se dibuje si llamamos a SpriteSheet::render(). Dado que este índice es un valor numérico, debemos conocer la posición dentro de la colección de la animación que queremos reproducir. Esto implica que cambiar la animación actual requeriría saber qué fila corresponde a la animación en cada SpriteSheet distinto, lo cual puede ser poco intuitivo en cuanto comencemos a trabajar con SpriteSheets distintos.

Para resolver este problema la clase SpriteSheet utiliza un mecanismo de vinculación mediante nombre, de forma que podemos vincular una posición en la colección con una cadena de caracteres.

Figura 4: Posicion de animaciones

De forma rudimentaria, si quisieramos cambiar la animacion actual de sprite deberámos hacer lo siguiente:

```
sprite.setAnimation(0);        // Animacion hacia arriba
sprite.setAnimation(3);        // Animacion hacia la izquierda
```

Una forma más sencilla de trabajar con las animaciones es utilizando el método bindAnimation() de la clase SpriteSheet de la siguiente forma:

```
SpriteSheet sprite = new SpriteSheet("pacman.bmp");

sprite.bindAnimation(0,"UP");     // Vincula la animacion 0 con el nombre UP
sprite.bindAnimation(1,"RIGHT"); // Vincula la animacion 1 con el nombre RIGHT
sprite.bindAnimation(2,"DOWN");  // Vincula la animacion 2 con el nombre DOWN
sprite.bindAnimation(3,"LEFT");  // Vincula la animacion 3 con el nombre LEFT
sprite.bindAnimation(4,"DIE");   // Vincula la animacion 4 con el nombre DIE

sprite.setAnimation("RIGHT");     // Cambia la animacion a la vinculada con RIGHT
```

Una vez la vinculación se ha hecho se puede trabajar con las animaciones mediante su nombre en lugar de su posición.

4.3.2. Velocidad de animación

La clase SpriteSheet también nos permite ajustar la velocidad a la que se reproducen las animaciones. Por ejemplo, en el SpriteSheet de pacman anterior, la animación de la muerte se reproduce más rápido que el resto. Para ello haremos uso de distintos valores de frame skip de la siguiente manera (supondremos que se han vinculado las animaciones usando el código anterior):

```
sprite.setFrameSkip(15);       // Aplica el frame skip a todas
sprite.setFrameSkip("DIE",5); // Frame skip especifico a DIE
```

Como vemos tenemos dos métodos, uno para aplicar un valor global y otro para animaciones específicas. Destacar que cuanto mayor sea el valor de frame skip más lenta será la animación, ya que el valor de frame skip indica cuantos renderizados tarda el sprite en pasar al siguiente frame de la animación.

5. Audio

Nuestra aplicación necesita reproducir sonidos para cuando el Pacman consigue una de las monedas y también para reproducir una canción de fondo. Para ello encapsularemos las funcionales de la biblioteca SDL_Mixer en dos clases distintas, una para sonidos y otra para música, ya que SDL_Mixer trabaja de forma distinta los sonidos y la música.

5.1. Sound

A la hora de reproducir un sonido con SDL_Mixer este le asigna un canal en el que reproducirse, de forma que a la hora de comprobar si el sonido se está reproduciendo o no lo haremos sobre el canal. Por ello los atributos de la clase serán:

```
Mix_Chunk*      _sound;
int             _channel;
```

Donde en _sound guardaremos la información sobre el archivo de sonido a reproducir y en _channel el canal donde se reproduce el sonido.

Para cargar un sonido desde un archivo hacemos uso de la siguiente llamada desde el mismo constructor:

```
this->_sound = Mix_LoadWAV("sonido.wav");
```

Una vez cargada la información del archivo podemos reproducirlo mediante el método play:

```
bool Sound::play()
{
    int isPlaying;
    // Comprobar si se esta reproduciendo algo en el canal
    isPlaying = Mix_Playing(this->_channel);

    if (isPlaying != 0) {        // Si no es 0, ya se esta reproduciendo
        return false;
    } else {
        this->_channel = Mix_PlayChannel(-1, this->_sound, 0);
    }
    return true;
}
```

Los atributos que se le pasan son -1 para que escojas el canal en el que reproducir automáticamente, la información del sonido y por último el número de veces que queremos que se repita el sonido (en este caso no queremos que se repita). Si quisieramos que se reproduciera en un bucle tan solo habría que pasarle -1.

5.2. Music

En el caso de la música funciona de forma similar a los sonidos pero SDL trabaja con ella de forma distinta, además de poder reproducir solo un canal a la vez, por lo que ya no necesitamos guardar información sobre el canal, por lo que los atributos serán tan solo:

```
Mix_Music*    _music;
```

Y cargaremos la inforación desde el constructor como sigue:

```
this->_music = Mix_LoadMUS("musica.ogg");
```

Esta vez queremos que la música se reproduzca en bucle, por lo que usaremos el siguiente método:

```
void Music::playLoop()
{
    if (Mix_PlayingMusic()) {
        std::cout << "La musica ya se esta reproduciendo" << std::endl;
    } else {
        Mix_PlayMusic(this->_music, -1);
    }
}
```

6. Lógica

Aquí podríamos hablar sobre varias clases como Entity y Enemy, pero se trata solo de modelos que contienen la posición de cada objeto y métodos para trabajar con ellos; moverlos, asignarles un sprite etc. Una clase que sí es interesante comentar es CollisionBox, que se usa ampliamente para comprobar colisiones entre objetos del juego.

6.1. CollisionBox

Para poder hacer que el jugador solo se pueda mover por los pasillos del escenario, sin poder atravesar los muros, o que el jugador recolecte las monedas, necesitamos una forma de saber cuando dos elementos chocan.

Se pueden implementar varios sistemas, siendo uno de los más sencillos el que usa esferas que envuelven los elementos y que comprueba si dos esferas colisionan mediante la distancia entre las dos esferas. En nuestro caso hemos escogido el uso de rectngulos, ya que es el siguiente sistema ms sencillo, se adapta bien a la forma del escenario y es más interesante de comentar.

Nuestras cajas de colisión estarán relacionadas con una posición concreta que se especifica cuando se crean, de forma que la caja siga a esa posición cuando esta cambien. De esta forma no necesitamos actualizar a mano la posición de la caja.

A la hora de comprobar si una caja colisiona con otra lo que haremos será buscar los casos en los que sabemos que no están colisionando, de forma que en cuanto se cumpla uno de ellos se deja devuelve false y no hay más que comprobar. De esta forma ahorramos muchos cálculos por frame teniendo en cuenta que se realizan muchas comprobaciones de colisiones cada frame.

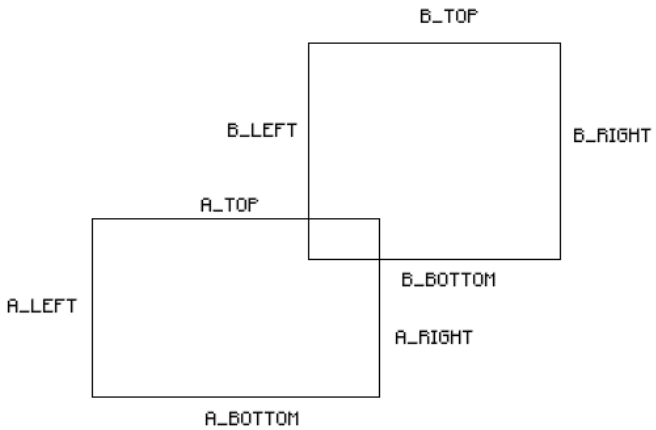

Figura 5: Cajas de colisión

Básicamente lo que haremos es, teniendo las dos cajas de la figura, A y B, buscar las situaciones que con una sola comprobación nos permitan descartar la colisión, esto es:

1. A_RIGHT $\leq B\_LEFT$.
2. A_TOP $\leq B\_BOTTOM$.
3. A_BOTTOM $\geq B\_TOP$.
4. A_LEFT $\geq B\_RIGHT$.

En código se traduce como:

```
bool CollisionBox::collides(CollisionBox &other)
{
    if (!((this->_boxPosition.x() + this->_width) >= other._boxPosition.x())) {
        return false;
    }

    if (!(this->_boxPosition.x() <= (other._boxPosition.x() + other._width))) {
        return false;
    }

    if  (!((this->_boxPosition.y() - this->_height) <= other._boxPosition.y())) {
        return false;
    }

    if (!(this->_boxPosition.y() >= (other._boxPosition.y() - other._height))) {
        return false;
    }
    return true;
}
```