# Práctica 1: Arkanoid 1.0

Curso 2018-2019. Tecnología de la Programación de Videojuegos. UCM

Fecha de entrega: 7 de noviembre de 2018

El objetivo de esta práctica es implementar en C++/SDL una versión simplificada del conocido juego *Arkanoid*, creado por Taito en 1986, que a su vez está basado en los *Breakout* de Atari de los años 70 (véase `https://es.wikipedia.org/wiki/Arkanoid`). Se utilizará la librería SDL para manejar toda la entrada/salida del juego. A continuación se detallan las principales simplificaciones y diferencias que nuestro juego tendrá respecto al original:

- El juego acaba cuando no quedan bloques (el jugador gana) o bien cuando el jugador se queda sin vidas (el jugador pierde). No hay por tanto concepto de puntuación con las simplificaciones que ello conlleva, ni de paso de nivel. Se deja libertad respecto a la manera en la que se informa al usuario tanto del número de vidas restantes, como de si gana o pierde cuando el juego acaba (en consola, en ventana emergente o en la propia ventana SDL).
- No hay bloques blindados, es decir, todos los bloques se destruyen con un solo golpe. Tampoco hay premios ni monstruitos.
- No hay animaciones de ningún tipo, ni al destruir bloques ni al perder vidas. Además la velocidad de la bola es siempre la misma.
- No hay fondo ni sombras y el número de colores de los bloques se reduce a seis.

# Detalles de implementación

## Diseño de clases

A continuación se indican las clases y métodos que debes implementar obligatoriamente. A algunos métodos se les da un nombre específico para poder referirnos a ellos en otras partes del texto. Deberás implementar además los métodos (y posiblemente las clases) adicionales que consideres necesario para mejorar la claridad y reusabilidad del código. Cada clase se corresponderá con la definición de un módulo C++ con sus correspondientes ficheros .h y .cpp.

**Clase Texture:** Permite manejar texturas SDL. Contiene al menos un puntero a la textura SDL e información sobre su tamaño total y los tamaños de sus frames. Implementa métodos para construir/cargar la textura de un fichero, para dibujarla en la posición proporcionada, bien en su totalidad (método `render`) o bien uno de sus frames (método `renderFrame`), y para destruirla/liberarla. Se darán detalles en clase.

**Clase Vector2D:** Representa vectores o puntos en dos dimensiones y por tanto incluye dos atributos (`x` e `y`) de tipo `double`. Implementa, además de la constructora, métodos para consultar las componentes `x` e `y`, para normalizar el vector, y para la suma, resta, producto escalar de vectores y producto de un vector por un escalar. La suma, resta y productos deben implementarse como operadores.

**Clase Wall:** Representa a los muros del juego. Contiene al menos la posición (de la esquina superior izquierda) como `Vector2D`, el ancho y el alto, y un puntero a su textura. Implementa la constructura y el método `render` para dibujarse.

**Clase Block:** Representa a los bloques del juego. Contiene al menos la posición absoluta (esquina superior izquierda) como `Vector2D`, el ancho y el alto, el color (entero para encontrar el frame de la textura), la fila y columna del bloque en el mapa de bloques, y un puntero a la textura con los bloques. Implementa la constructura y el método `render` para dibujarse.

**Clase BlocksMap:** Contiene el mapa de bloques del juego representado como una matriz dinámica de punteros a bloques (tipo `Block`) con sus dimensiones asociadas, y los tamaños en píxeles del mapa y de cada celda. Implementa, además de la constructora y destructora, métodos para cargarse de fichero, dibujarse (`render`), consultar el número de bloques, y determinar el bloque (`Block*`) y el vector de colisión de la pelota (en caso de haber colisión con ella).

**Clase Paddle:** Representa a la plataforma del juego. Contiene al menos la posición (esquina superior izquierda) como `Vector2D`, el ancho y el alto, la dirección/velocidad (`Vector2D`), y un puntero a su textura. Implementa, además de la constructura, métodos para dibujarse (`render`), actualizarse, moviéndose acorde a su dirección/velocidad (`update`), tratar los eventos (`handleEvents`) que controlan el movimiento (cursores izquierdo y derecho) y determinar el vector de colisión con la pelota (en caso de haberla). Observa que el ángulo con el que rebota la pelota depende también del punto de la plataforma en el que la pelota rebota. Se darán detalles en clase.

**Clase Ball:** Representa a la pelota del juego. Contiene al menos la posición (esquina superior izquierda) como `Vector2D`, el ancho y el alto, la dirección/velocidad (`Vector2D`), un puntero a su textura, y un puntero al juego (para que éste le diga, mediante su método `collides`, si colisiona con algún objeto, y con qué vector de colisión). Implementa, además de la constructura, métodos para dibujarse (`render`) y actualizarse (`update`), moviéndose acorde a su dirección/velocidad o cambiando de dirección de acuerdo al vector de colisión en caso de rebotar contra algún objeto.

**Clase Game:** Contiene, al menos, punteros a la ventana y al *renderer* de SDL, booleanos de control del juego (`exit`, `gameover`, `win`, ...), un array con los punteros a todas las texturas del juego, y los (punteros a los) objetos del juego. Define también las constantes que sean necesarias (dimensiones de la ventana y de los objetos del juego, `FRAME_RATE` para controlar la velocidad del juego, etc.). Implementa métodos para inicializarse y destruirse, el método `run` con el bucle principal del juego, métodos para dibujar el estado actual del juego (método `render`), actualizar (método `update`), manejar eventos (método `handleEvents`), y para determinar si la pelota colisiona con algún objeto, y en tal caso, con qué vector de colisión (método `collides`).

## Formato de ficheros de mapas

El mapa de bloques se lee de un fichero de texto con el siguiente formato: En la primera línea hay dos números enteros separados por un espacio que indican el número de filas y columnas respectivamente del mapa. A continuación viene el contenido del mapa. Cada fila del mapa se representa en el fichero en una línea que contiene números enteros en el rango $0-6$ separados mediante espacios. Cada número representa el contenido de la celda correspondiente a la posición en la que se encuentra, siendo un 0 un hueco y cualquier otro número un bloque del color dado por el valor (se deja libertad en la asignación de valores para los colores elegidos). Ten en cuenta que el mapa debe escalarse al espacio correspondiente de la ventana reservado para él.

### Carga de texturas

Las texturas con las imágenes del juego deben cargarse durante la inicialización del juego (en la constructora de `Game`) y guardarse en un array (de `NUM_TEXTURES` elementos de tipo `Texture*`) cuyos índices serán valores de un tipo enumerado (`TextureName`) con los nombres de las distintas texturas. Los nombres de los ficheros de imágenes y los números de frames en horizontal y vertical, deben estar definidos (mediante inicialización homogénea) en un array constante de `NUM_TEXTURES` estructuras en `Game.h`. Esto permite automatizar el proceso de carga de texturas.

### Renderizado del juego

En el bucle principal del juego (método `run`) se invoca al método del juego `render`, el cual simplemente delega el renderizado a los diferentes objetos del juego (muros, mapa de bloques, pelota y plataforma) llamando a sus respectivos métodos `render`, y finalmente presenta la escena en la pantalla (llamada a la función `SDL_RenderPresent`). Cada objeto del juego sabe cómo pintarse, en concreto, conoce su posición y tamaño, y contiene un puntero a su textura asociada. Por lo tanto, construirá el rectángulo de destino e invocará al método `render` o `renderFrame` de la textura correspondiente, quién realizará el renderizado real (llamada a la función `SDL_RenderCopy`).

### Manejo básico de errores

Debes usar excepciones de tipo `string` (con un mensaje informativo del error correspondiente) para los errores básicos que pueda haber. En concreto, es obligatorio contemplar los siguientes tipos de errores: fichero de imagen no encontrado o no válido, fichero de mapa de bloques no encontrado o no válido, y error de SDL. Puesto que en principio son todos ellos errores irrecuperables, la excepción correspondiente llegará hasta el `main`, donde deberá capturarse, informando al usuario con el mensaje de la excepción antes de cerrar la aplicación.

### Movimientos y manejo de colisiones de la pelota

Los movimientos de los objetos del juego los desencadena el juego y se van delegando de la siguiente forma: el método `update` del juego (invocado desde el bucle principal del método `run`), va llamando a los métodos `update` de cada uno de los elementos móviles del juego, en este caso, pelota y plataforma. Son ellos los que conocen dónde se encuentran y cómo deben moverse. Para saber si un movimiento concreto en una dirección se puede realizar o no, deben conocer su *entorno*, es decir, deben preguntar al juego, bien consultando alguna constante o bien llamando a métodos. En particular, la pelota preguntará al juego (llamando al método `collides`) si en la dirección en la que se mueve, colisiona o no con algún objeto, y en tal caso con qué vector de colisión. Con esa información la pelota actualizará su vector de dirección. Dados el vector de dirección de la pelota $D$ y el vector normalizado de colisión $C$, el vector de reflexión $R$ de la pelota se calcula como $R = D - 2 * (A * C) * C$.

## Pautas generales obligatorias

A continuación se indican algunas pautas generales que vuestro código debe seguir:

- Asegúrate de que el programa no deje basura. Para que Visual te informe debes escribir al principio de la función `main` esta instrucción

  ```
  _CrtSetDbgFlag( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
  ```

  Para obtener información más detallada y legible debes incluir en todos los módulos el fichero `checkML.h` (disponible en la plantilla en el proyecto `HolaSDL`).

- Todos los atributos deben ser privados excepto quizás algunas constantes del juego en caso de que se definan como atributos estáticos.
- Define las constantes que sean necesarias. En general, no deben aparecer literales que pudiesen corresponder con configuraciones del programa en el código.
- No debe haber métodos que superen las 30-40 líneas de código.
- Escribe comentarios en el código, al menos uno por cada método que explique de forma clara qué hace el método. Sé cuidadoso también con los nombres que eliges para variables, parámetros, atributos y métodos. Es importante que denoten realmente lo que son o hacen. Preferiblemente usa nombres en inglés.

## Funcionalidades opcionales (2 puntos adicionales máximo)

- Extender la mecánica del juego para que al romper todos los bloques se pase a otra pantalla.
- Mostrar en la escena un contador en el que se muestre el tiempo transcurrido en la partida. Para ello debes definir una nueva clase con su textura, posición y tamaño, y con los métodos `render` y `update`.
- Mantener un registro con los mejores 10 tiempos en haber acabado el juego en partidas pasadas y actualizarlo con cada partida acabada cuyo tiempo entre en el *top-10*. Ver el ejercicio 5 del tema 2.
- Implementar el soporte para permitir guardar y cargar partidas.

## Entrega

En la tarea del campus virtual *Entrega de la práctica 1* y dentro de la fecha límite (7 de noviembre), cada uno de los miembros del grupo, debe subir un fichero comprimido (.zip) que contenga los archivos de código .h y .cpp del proyecto, las imágenes que se utilicen, y, un archivo `info.txt` con los nombres de los componentes del grupo y unas líneas explicando las funcionalidades opcionales incluidas y/o las cosas que no estén funcionando correctamente. Además, para que la práctica se considere entregada, deberá pasarse una *entrevista* en la que el profesor comprobará, con los dos autores de la práctica, su funcionamiento en ejecución, y si es correcto realizará preguntas (individuales) sobre la implementación. Las entrevistas se realizarán en las dos sesiones de laboratorio siguientes a la fecha de entrega, o si fuese necesario en horario de tutorías.

**Entrega intermedia el 23 de Octubre:** Un 20 % de la nota se obtendrá mediante una entrega intermedia, que tendrá lugar en la sesión de laboratorio del día 23 de octubre, en la que el profesor revisará el estado actual de vuestra práctica. En particular, se espera que vuestra práctica cargue el mapa y visualice toda la escena, aún sin ningún movimiento.