

Práctica 1: Super Mario 1.0

Curso 2024-2025. Tecnología de la Programación de Videojuegos 1. UCM

Fecha de entrega: 5 de noviembre de 2023

El objetivo de esta práctica es implementar en C++/SDL una versión simplificada del conocido juego *Super Mario Bros.* creado por Shigeru Miyamoto (宮本 茂). Se utilizará la biblioteca SDL para manejar la entrada/salida del juego. Tomaremos como referencia la versión original para NES de 1985, que se puede jugar online en supermarioplay.com. A continuación se detallan las simplificaciones y diferencias principales que nuestro juego tendrá respecto al original:



- El juego solo tiene un nivel, en el que aparecen *goombas* y *koopas* como enemigos y *superchampiñones* como potenciadores. 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).
- Se obvian todas las animaciones de los objetos del juego salvo la variación de aspecto en el movimiento de los personajes y la alternancia de color de los bloques sorpresa. Los bloques no se mueven al ser golpeados.
- No se controla el tiempo restante ni la puntuación.

En futuras versiones de la práctica se podrán recuperar algunas de estas características.

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 (salvo la plantilla **Vector2D**) se corresponderá con un par de archivos `.h` y `.cpp`.


Clase Texture (disponible en el CV): encapsula el manejo de las texturas SDL. Contiene un puntero a la textura SDL e información sobre su tamaño total y el tamaño de sus frames. Esta clase 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.


Clase Vector2D: es un tipo plantilla que representa vectores o puntos en dos dimensiones y por tanto incluye dos atributos (**x** e **y**) de un tipo genérico **T**. Implementa, además de la constructora, métodos para consultar las componentes **x** e **y**, 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. En el mismo módulo define con **using** un alias **Point2D** para el tipo.


Clase TileMap: se encarga de dibujar el fondo y los obstáculos del nivel según avanza el personaje. Contiene un puntero a la textura del conjunto de patrones y una matriz bidimensional de índices para describir el mapa (el funcionamiento de esta clase se explica en una sección posterior). Implementa un constructor que lee dicha matriz desde un archivo CSV y métodos para dibujarse (**render**), actualizarse (**update**, si procede) y detectar colisiones sobre los obstáculos (**hit**).

Clase Block: contiene la posición de un bloque (tipo `Point2D`), el tipo de bloque (ladrillo  sorpresa , vacío  u oculto) como un enumerado, una acción (*potenciador* o *moneda*) también como un enumerado, un puntero a su textura y un puntero al juego. Implementa un constructor, métodos para dibujarse (`render`), actualizarse (`update`) y manejar colisiones (`hit`). El bloque será un obstáculo para todos los personajes cuando lo golpeen desde arriba o desde un lateral, pero cuando Mario lo golpee desde abajo ocurrirá lo siguiente: si el bloque es de ladrillo y el protagonista es Super Mario, el bloque se destruirá; si el bloque es sorpresa u oculto y la acción es *potenciador*, hará aparecer una superchampiñón sobre el bloque y lo convertirá en bloque vacío; y no hará nada en el resto de casos.

Clase Goomba (): contiene al menos la posición actual del goomba (tipo `Point2D`), su dirección de movimiento, un puntero a su textura y un puntero al juego. Más abajo se explica el movimiento de los personajes. Implementa además métodos para construirse, dibujarse (método `render`), actualizarse (método `update`) y manejar colisiones (`hit`). Si el goomba es golpeado desde arriba por el jugador morirá, pero con cualquier otra colisión será el jugador el que pierda una vida.

Clase Koopa (): representa un Koopa Troopa básico del juego original y tiene los mismos atributos y comportamiento que el goomba. Sin embargo, en futuras versiones haremos que al ser golpeado se convierta en un caparazón y pueda ser lanzado contra los enemigos.

Clase Mushroom (): representa un superchampiñón que se mueve inicialmente hacia la derecha en el mapa, cambiando de dirección cuando choca con un obstáculo. Si Mario colisiona con el superchampiñón se convierte en Super Mario. Contiene una posición (tipo `Point2D`), una dirección de movimiento, un puntero a su textura y un puntero al juego.

Clase Player (): al igual que los anteriores, sus atributos incluyen su posición actual, un puntero a su textura y un puntero al juego. Además, mantiene la dirección actual del movimiento, el número de vidas que le quedan (inicialmente 3) y su aspecto (Mario o Super Mario). Implementa también métodos para construirse, dibujarse (`render`), actualizarse, es decir, moverse (método `update`), recibir daño (método `hit`) y manejar eventos del teclado (método `handleEvent`), que determinan el estado de movimiento.

Cuando Mario colisione con un enemigo o caiga en una fosa del mapa se reducirá su número de vidas y volverá a su posición inicial en el nivel. Super Mario, en cambio, al chocar con un enemigo se convertirá en Mario, permanecerá donde está y se mantendrá invulnerable durante un par de segundos. La partida finaliza cuando el jugador pierda todas sus vidas o alcance el mástil del final del nivel.

Clase Game: contiene, al menos, el tamaño de la ventana, el desplazamiento actual del mapa (`mapOffset`), punteros a la ventana y al `renderer`, a los elementos del juego (con el tipo `vector`), el booleano `exit`, y el array de texturas (ver más abajo). Define también las constantes que sean necesarias. Implementa métodos públicos 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`) y manejar eventos (método `handleEvents`). Además, sin abusar, tendrá otros métodos auxiliares que hagan falta como `getMapOffset` o `collides` (ver más abajo).

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 estático (tipo `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.cpp`. Esto permite automatizar el proceso de carga de texturas. Utiliza una constante auxiliar `TEXTURE_ROOT` con valor `"../assets/images/"` para evitar repetir la parte común de la ruta en cada entrada del array de texturas.

Renderizado del juego

En el bucle principal del juego (método `run`) se invoca al método del juego `render`, que simplemente delega el renderizado a los diferentes objetos del juego 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, tamaño, y textura asociada. Por lo tanto, cada cual construirá su rectángulo de destino e invocará al método `render` o `renderFrame` de la textura correspondiente, que realizará el renderizado real (llamada a la función `SDL_RenderCopy`). Como la vista del mapa se va desplazando con el avance del protagonista es importante distinguir el rectángulo de colisión (en coordenadas absolutas) del rectángulo de renderizado (en coordenadas relativas a la vista).

Movimiento de los personajes

Los movimientos de los personajes del juego se organizan desde la clase `Game` 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 del juego. Son ellos los que conocen dónde se encuentran y cómo deben moverse. Los goombas, koopas y superchampiñones se mueven en una dirección fija, caen si dejan de tener un obstáculo a sus pies, invierten su dirección si encuentran un obstáculo horizontal y desaparecen si caen por un hueco bajo el suelo del mapa. Esta interacción requiere que los objetos del juego tengan un puntero a `Game` para llamar a su método `collides`.

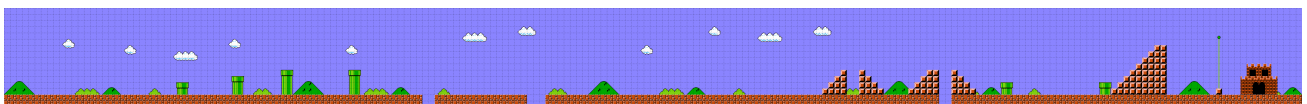
El movimiento del protagonista está controlado por el usuario a través del teclado. Mario se desplazará en horizontal a velocidad constante mientras se mantengan pulsadas las flechas izquierda o derecha (u otras teclas equivalentes) del teclado. Esta funcionalidad se implementará de la siguiente forma: cuando se pulsa o suelta una tecla de desplazamiento, el bucle de manejo de eventos (en el método `handleEvents`) delega el manejo de este evento en el objeto del jugador (mediante invocación a su método `handleEvent`), que actualizará su dirección de movimiento en consecuencia (izquierda, derecha o quieto). El protagonista (en su método `update`) se desplazará siguiendo la dirección establecida. Es importante no realizar la actualización de la posición directamente al pulsar la tecla, sino a través de la dirección, porque si no Mario se desplazará a trompicones. Si se pulsa la barra espaciadora y está apoyado sobre un obstáculo, Mario saltará hasta alcanzar una determinada altura o colisionar con un objeto, momento en el que empezará a caer.

Eliminación de los objetos del juego

Los personajes del juego desaparecen de la escena al ser atacados, al atravesar el borde izquierdo del mapa o al caer por su parte inferior. Al finalizar la etapa de actualización, `Game` revisará la colección de objetos de la escena eliminando aquellos que hayan desaparecido. Cada una de las clases del juego tendrá un método `isAlive` que devolverá un valor booleano indicando si el objeto sigue vivo o ha de ser eliminado por `Game`, liberando la memoria que este ocupase.

Formato de ficheros de mapas y configuraciones iniciales

El mapa de un nivel del juego se compone de (1) un mosaico de 16 celdas de alto y tantas de ancho como sean necesarias para el fondo y los objetos estáticos y (2) el resto de personajes y objetos interactivos del mapa. Cada componente se describe con un archivo de texto, `worldX.csv` y `worldX.txt`, respectivamente, donde `worldX` es el nombre del mapa.



El mosaico estático se describe mediante una tabla de índices en **formato CSV** sobre el conjunto de patrones que se muestra a continuación. Cada línea del archivo CSV contiene una secuencia de números enteros separados por comas que representan una fila completa del nivel. Las celdas vacías se indican con un -1 y han de mostrar el color de fondo (138, 132, 255).

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62

Los patrones de la 4 primeras columnas son obstáculos y el resto son decoraciones del fondo.

Por otro lado, el archivo de objetos dinámicos `worldX.txt` introduce en cada línea un elemento distinto. La línea comienza con un identificador del tipo de objeto (**M** para Mario, **B** para un bloque, **G** para un goomba, **K** para un koopa y **H** para un superchampiñón), seguido de su posición y finalmente de los atributos extra de cada personaje, separados por espacios. La posición será un par de números cuyas unidades son las celda del mosaico del mapa desde la esquina superior izquierda. El objeto aparecerá centrado horizontalmente y apoyado sobre la base de su celda. Los objetos que tiene atributos extra son: el jugador, con el número de vidas; y el bloque, con un carácter para su tipo (**B**=ladrillo, **?**=pregunta, **0**=vacío, **H**=oculto) y, si el bloque es de tipo pregunta u oculto, la acción que realiza (**P**=potenciador, **C**=moneda).

Cada objeto del juego dispondrá de un constructor que recibe un argumento de tipo `std::istream&` del que leer sus propios datos (salvo el identificador, que se leerá en `Game` para llamar al constructor adecuado). Para facilitar la identificación de problemas de lectura y la compatibilidad entre versiones, el método `LoadMap` de `Game` leerá líneas completas con `getline` y construirá un flujo temporal `istringstream` para pasárselo al constructor del objeto (se darán más detalles en clase). Las líneas vacías o que comiencen con una almohadilla se ignorarán.

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 del juego no encontrado, y error de SDL. Puesto que en principio son todos ellos errores irreversibles, la excepción correspondiente llegará hasta el `main`, donde deberá capturarse, informando al usuario con el mensaje de la excepción antes de cerrar la aplicación. Recuerda que el tipo del literal `"error"` no es `string` sino `const char*` y has de escribir `string("error")` o `"error"s` en el `throw` para que tenga el tipo adecuado.

Pautas generales obligatorias

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

- Asegúrate de que el programa no deje basura. La plantilla de Visual Studio incluye el archivo `checkML.h` que debes introducir como primera inclusión en todos los archivos de implementación.
- Todos los atributos deben ser privados excepto quizás algunas constantes del juego definidas 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

1. Extender la mecánica del juego para llevar cuenta de la puntuación obtenida. En el juego original se obtienen 100 puntos por espachurrar un goomba o koopa, 200 puntos por golpear un bloque de pregunta, 50 por destruir un bloque de ladrillo y 1000 por capturar un superchampiñón. La puntuación actual del juego se puede mostrar por consola cuando se consigue nueva puntuación.
2. Implementar un nuevo objeto del juego **Shell** () que aparecerá en el mapa cuando se haya vencido a un koopa. Si Mario salta sobre él, el caparazón saldrá disparado en la dirección contraria al extremo sobre el que haya saltado eliminando a todos los enemigos (o amigos) que encuentre a su paso y rebotando en los obstáculos hasta salir de la vista por la izquierda o por abajo. Si se implementa la funcionalidad de puntuación, saltar sobre él añadirá 500 puntos a la puntuación del jugador.
3. Implementar una clase **InfoBar** cuyo método **render** se encargue de mostrar en la ventana SDL una barra de información del juego que incluya al menos el número de vidas restantes y posiblemente también la puntuación actual.
4. Extendiendo el formato de los mapas iniciales, implementar el soporte para guardar y cargar partidas.

Entrega

En la tarea *Entrega de la práctica 1* del campus virtual y dentro de la fecha límite (ver junto al título), cualquiera de los miembros del grupo debe subir el fichero comprimido (.zip) generado al ejecutar en la carpeta de la solución un programa que se proporcionará en la tarea de la entrega.

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 (posiblemente individuales) sobre la implementación. Se darán detalles más adelante sobre las fechas, forma y organización de las entrevistas.

Entrega intermedia el 29 de octubre: un 20% de la nota se obtendrá el día 29 de octubre en función de vuestros avances. Para ello debéis mostrar el funcionamiento de vuestra práctica ese mismo día en la sesión de laboratorio. Para obtener la máxima nota en esta entrega se espera que vuestra práctica muestre el mapa original y mueva a Mario.