

# Práctica 2: Super Mario 2.0

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

Fecha de entrega: 26 de noviembre de 2024

El objetivo fundamental de esta práctica es incorporar la herencia y el polimorfismo en la programación de videojuegos mediante C++/SDL. Para ello, partiremos de la práctica anterior y desarrollaremos una serie de mejoras que se explican a continuación. En cuanto a funcionalidad, el juego presenta las siguientes modificaciones o extensiones, siendo la segunda una funcionalidad opcional de la práctica anterior que ahora es obligatoria:

1. Como en el juego original, al finalizar un nivel se pasa al nivel siguiente. Se proporciona un mundo más, que se deberá cargar cuando el jugador complete el nivel 1 (sin las animaciones del original).
2. El jugador obtiene puntos cada vez que un enemigo es destruido o al golpear los bloques. 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 acumulada se muestra en la terminal o continuamente en la ventana del juego.
3. Se introducen nuevos objetos del juego: plataformas, monedas y opcionalmente plantas piraña.

## Detalles de implementación

### Diseño de clases

A continuación se indican las modificaciones con respecto la práctica anterior que has de implementar obligatoriamente. Añade además los métodos (y posiblemente las clases) adicionales que consideres necesario para mejorar la claridad y reusabilidad del código. Como norma general, cada clase se corresponderá con la definición de un par de archivos .h y .cpp. Las nuevas clases de esta práctica son las siguientes:

**Clase GameObject:** la clase abstracta `GameObject` es la raíz de la jerarquía de objetos del juego y reúne la funcionalidad común a todos ellos. Su declaración incluye los métodos virtuales puros `render` y `update`, además de su destructora virtual y un atributo con un puntero al juego.

**Clase SceneObject:** es una subclase todavía abstracta de `GameObject` de la que descienden todos los personajes de la escena (es decir, todas las clases obligatorias de la práctica anterior). Mantiene la posición del objeto en la ventana, sus dimensiones y su velocidad como atributos. Declara un método virtual `hit` que recibe como argumento el rectángulo de ataque y si el ataque proviene del jugador. Las subclases deberán implementar este método según su naturaleza. Guardará además un ancla a la lista de objetos de la escena para facilitar su eliminación cuando corresponda (véase más abajo).

**Clase Enemy:** es la clase común de todos los enemigos del juego, incluidas las clases `Goomba` y `Koopa` de la práctica anterior, que ahora deberán heredar de ella. Todos estos enemigos tiene un comportamiento uniforme respecto a sus colisiones con Mario que se debe implementar en el método `hit` de esta clase.

**Clase Pickable:** es la clase común de todos los objetos capturables por Mario, incluida la clase `Mushroom` de la práctica anterior y la nueva clase `Coin`. El comportamiento en caso de colisión con Mario es también muy semejante para todos los objetos capturables y se habrá de implementar en esta clase, delegando en las subclases para el efecto concreto que desencadenan con un método virtual puro protegido `triggerAction`.

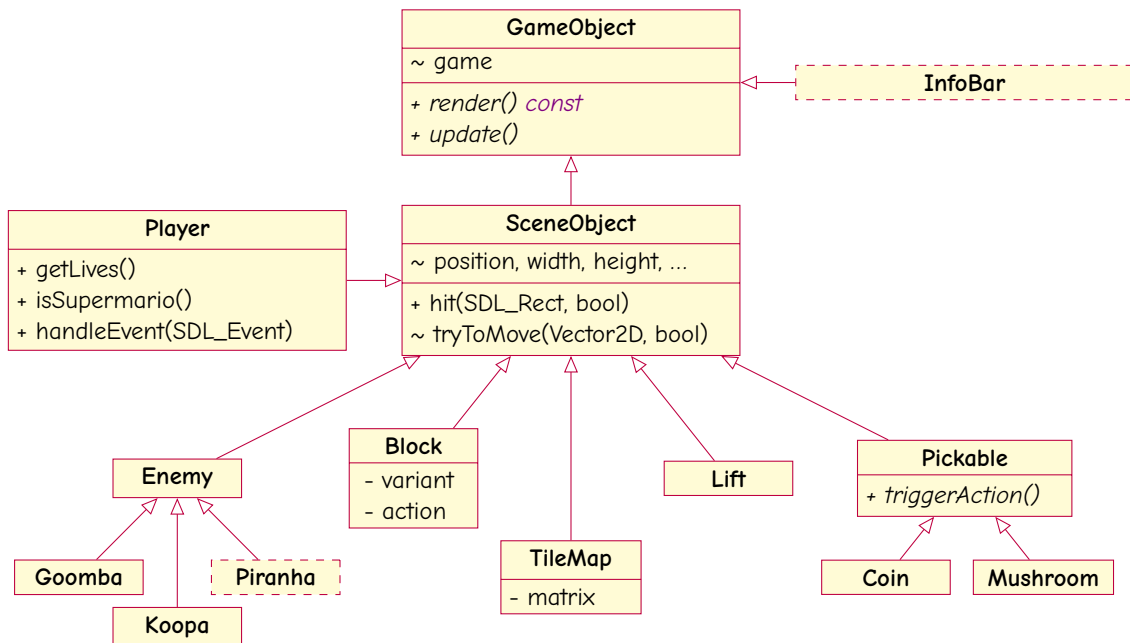


Figura 1: Digrama de clases principales del juego.

**Clase Coin (🪙):** representa una moneda del juego que aporta 200 puntos al jugador cuando la captura. Es una subclase de **Pickable** e implementa el método **triggerAction** para añadir los puntos al juego a través de un posible método **givePoints** de **Game**.

**Clase Lift (🚢):** es un ascensor que se mueve en bucle a velocidad constante en vertical, reapareciendo por un extremo cuando se oculta por el opuesto. Los personajes se pueden subir a él para salvar obstáculos, aunque morirán si descienden o ascienden fuera de la vista. Es una subclase **SceneObject**, no causa daño directo y se representa en el archivo del mapa con un argumento extra para su velocidad vertical.

Las clases de la práctica 1 se han de adaptar de forma natural para hacerlas subclases de sus respectivas madres como aparece reflejado en la figura 1. Algunas de ellas necesitan cambios adicionales:

**Clase Game:** en lugar de almacenar múltiples vectores para cada tipo de objeto del juego y tratarlos específicamente, utilizará una lista polimórfica de punteros a **SceneObject**. Los métodos **update**, **render** y demás solo tendrán que aplicar la misma operación a todos los objetos de la lista. Los objetos de tipo **GameObject** que no sean **SceneObject** (tal vez **InfoBar**) se pueden manejar por separado por simplicidad.

## Lista de objetos de la escena y su eliminación

El almacenamiento de los objetos del juego se ha implementado en la primera práctica con punteros sueltos y vectores de punteros para cada tipo de objeto. El polimorfismo que ahora conocemos permite simplificar esto enormemente, pues basta tener un solo contenedor con los punteros a todos los objetos a través de la clase madre **SceneObject**. El tipo concreto de los objetos es irrelevante (y no debemos hacer nada para averiguarlo) pues es suficiente aplicar polimórficamente los métodos virtuales **update**, **render** y **hit** declarados en la clase general. Así pues, podríamos utilizar un atributo de tipo **vector<SceneObject\*>** para almacenar los objetos del juego, pero el tipo **vector** no es ideal para almacenar un conjunto de objetos en el que unos aparecen y otros desaparecen continuamente. Una lista doblemente enlazada es una estructura de datos más conveniente en este contexto.

El material adjunto incluye la implementación completa de una clase **GameList** (archivo **gameList.h**) que pretende facilitar el manejo de los objetos del juego. La interfaz de esta clase (y su implementación) es semejante a la de una lista de la STL, se puede iterar sobre sus elementos como de costumbre y añadir elementos nuevos con **push\_back** y **push\_front**.

```

template <typename T>
class GameList {
public:

```

```

void push_back(T* elem);
void push_front(T* elem);
bool empty() const;

auto begin();
auto end();

class anchor { /* ... */ }; /* ... */
};

```

Sin embargo, esta clase impone también obligaciones al tipo `T`, que este caso será `SceneObject`. Con el fin de que los objetos se puedan autoeliminar de forma segura y eficiente, cada objeto guardará una especie de iterador (de tipo `anchor`) a su nodo en la lista enlazada. Este iterador lo fijará la propia clase `GameList` en sus métodos `push_back` y `push_front`, para lo que será necesario declarar un atributo `anchor` de tipo `GameList<SceneObject>::anchor` en `SceneObject` y un método público `setListAnchor` definido como

```

void setListAnchor(GameList<SceneObject>::anchor& anchor) {
    this->anchor = std::move(anchor);
}

```

Obsérvese que se está moviendo el argumento al atributo `anchor` de `SceneObject`. Cuando el objeto `SceneObject` se destruya, siguiendo la secuencia natural de eliminación de los objetos, se destruirá su atributo `anchor` y esto implicará automáticamente su eliminación de la lista. Es seguro eliminar un objeto mientras se está iterando sobre él, así que el objeto se puede eliminar en la propia clase con `delete this`.

## Ataques y colisiones entre objetos

Como en la práctica anterior, cuando un objeto se mueva llamará desde su método `update` al método `checkCollision` de `Game` para comprobar si hay algún obstáculo en su camino. El método `checkCollision` a su vez llamará a los métodos `hit` de todos los objetos de la escena, comprobará si se produce una colisión con alguno e informará al llamante del resultado. En el movimiento de los objetos capturables y los enemigos solo se informará de colisiones que impidan el movimiento (el método `hit` de `Player` no realizará ninguna acción y contestará que no hay colisión). Sin embargo, el movimiento del jugador (o del caparazón) sí que podrá causar daño a enemigos (en su método `hit`) o al propio jugador (al recibir el resultado en su `update`). También podrá desencadenar los efectos de los bloques o de los elementos capturables.

La clase `SceneObject` proporciona un método protegido `tryToMove` que recibe un vector velocidad e intenta aplicar dicho movimiento. Las subclases podrán utilizarlo para implementar su propio comportamiento. Se intentará el desplazamiento primero en vertical, comprobando si se producen colisiones y deteniendo el movimiento en tal caso, y a continuación en horizontal, volviendo a comprobar las colisiones.

Si la posición del objeto es  $(x, y)$  (esquina inferior izquierda), sus dimensiones  $(w, h)$  y su velocidad  $(u, v)$ , `tryToMove` comprobará en primer lugar la colisión del rectángulo  $[x, x + w] \times (y + v - h, y + v]$  con todos los objetos del juego a través de `checkCollision`. La posición se actualizará a  $y' := y + v - h_1$  donde  $h_1$  es la altura del objeto original que ha atravesado el otro objeto o cero si no hay colisión. A continuación se comprobará la colisión del rectángulo  $[x + u, x + u + w] \times (y' - h, y']$  con `checkCollision` y se actualizará  $x' := x + u - w_2$  con  $w_2$  la anchura de la intersección.

*Se completará esta sección con más detalles.*

## Cambios en el formato de los mapas

El formato de los mapas se mantiene prácticamente inalterado respecto a la práctica anterior, con el par de archivos `worldn.csv` y `worldn.txt`. Solo hay un pequeño cambio en el segundo archivo: su primera línea contiene ahora 3 números con los valores RGB del color de fondo del nivel (en el mundo 1 era azul, pero en el mundo 2 es negro) y la enumeración de los objetos del juego comienza en la línea siguiente.

```

138 132 255 } → RGB
M 1 14 3
...

```

El formato se extiende además para representar los nuevos objetos del juego: **L** para un ascensor (con su velocidad vertical como argumento extra), **C** para una moneda y **P** para una planta piraña (opcional). El archivo también puede incluir pseudobjetos **X** que representan puntos de control del nivel (funcionalidad opcional).

## Reinicios del nivel e inserción diferida de objetos

La lectura de objetos del mapa presenta la complicación de que estos han de quedar inactivos hasta que entren en escena. En la práctica anterior se sugirieron varias alternativas para conseguir este comportamiento (atributo **frozen**, listas separadas, contador de activos, etc.). Ahora, aprovechando el polimorfismo, se puede utilizar un vector **objectQueue** de tipo **vector<SceneObject\*>** sobre el que leer los objetos al cargar el mapa. Utilizando un contador se podrán extraer los objetos de esa cola conforme vayan haciéndose visibles y para evitar tener que recargar el mapa desde el archivo al reiniciar el nivel (porque Mario haya sido herido) se recomienda utilizar un método virtual puro

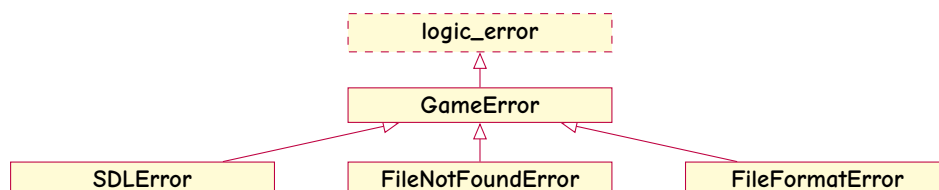
```
virtual SceneObject* clone() const = 0;
```

en **SceneObject** con el que poder clonar los objetos polimórficamente e insertar copias en la lista de objetos del juego.

Hay que tener en cuenta que cuando se reinicia el nivel todos los objetos del juego (salvo el jugador y el *tile map*) han de ser destruidos y reemplazados por objetos nuevos. Lo mismo ha de ocurrir si el reinicio se produce en un punto de control y no al principio del mapa.

## Jerarquía de excepciones

El manejo de los errores del juego se ha de implementar mediante las siguientes clases de excepciones. Las excepciones irreversibles se han de capturar en el **main** y la función **SDL\_ShowSimpleMessageBox** puede utilizarse para mostrar el mensaje de error al jugador:



**Game\_Error:** hereda de **std::logic\_error** y sirve como superclase de todas las demás excepciones que definiremos, proporcionando la funcionalidad común necesaria. Reutiliza el constructor y método **what** de **logic\_error** para el almacenamiento y uso del mensaje de la excepción.

**SDL\_Error:** hereda de **Game\_Error** y se utiliza para todos los errores relacionados con la inicialización y uso de SDL. Utiliza la función **SDL\_GetError** para obtener un mensaje específico sobre el error de SDL que se almacenará en la excepción.

**FileNotFound\_Error:** hereda de **Game\_Error** y se utiliza para los errores provocados al no encontrarse un fichero que el programa trata de abrir. El mensaje del error debe incluir el nombre del fichero en cuestión.

**FileFormat\_Error:** hereda de **Game\_Error** y se utiliza para los errores provocados en la lectura de los archivos de datos del juego (mapas). La excepción debe almacenar y mostrar el nombre de archivo y el número de línea del error junto con el mensaje.


## Pautas generales obligatorias

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

- Se debe hacer un buen uso de la herencia y del polimorfismo de manera que el código sea claro, se eviten las repeticiones de código, distinciones de casos innecesarias, no se abuse de castings ni de consultas de tipos en ejecución.

- 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 o protegidos 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. Añade un contador de tiempo máximo para completar el nivel como en el juego original. El tiempo disponible inicial o tras perder una vida son 400 segundos y si se agota el jugador perderá una vida como si hubiera sido atacado. El tiempo restante se mostrará en la ventana durante el juego y al finalizar el nivel se sumará a la puntuación obtenida.
2. Implementa 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 la puntuación actual.
3. Introduce la planta piraña () al juego como subclase `Piranha` de `Enemy`. La planta aparecerá periódicamente desde la tubería y causará daño al jugador si colisiona con ella (en el juego original no aparece si Mario está al lado o sobre la tubería). Será necesario añadir las correspondientes entradas en el archivo `world2.txt` del mapa, ya que no están incluidas por ser opcional.

P 103.5 13

P 109.5 12

P 115.5 14

4. Implementa el control de Mario mediante el ratón. La pulsación de cualquiera de los botones hará que salte y este se moverá a su velocidad habitual hacia la posición horizontal del ratón en la ventana.
5. Implementa la lógica de los puntos de control o *checkpoint*. Si el personaje rebasa la horizontal de un punto de control, cada vez que muera volverá a ese punto, en lugar de al inicio del mapa. El resto de objetos del juego volverán a su estado inicial.
6. Incorpora sonido al juego utilizando la biblioteca `SDL_mixer`. El juego original hace sonar su célebre melodía en bucle y acompaña de efectos sonoros las acciones del juego.

## Entrega

En la tarea *Entrega de la práctica 2* 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. La carpeta debe incluir 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 (posiblemente individuales) sobre la implementación. Se darán detalles más adelante sobre las fechas, forma y organización de las entrevistas.