

# Práctica 3: Super Mario 3.0

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

Fecha de entrega: 9 de diciembre de 2024 (12:00)

El objetivo fundamental de esta práctica es introducir una arquitectura escalable para el manejo de los estados de un juego. Para ello, partiendo del Super Mario 2.0 de la práctica anterior, extenderemos el juego con los siguientes nuevos estados:

- Al arrancar el programa aparecerá el *menú principal*, que permitirá iniciar una nueva partida desde el nivel 1 o desde el nivel 2 (si faltara algún mapa se informaría al activar esa opción, preferiblemente mediante `SDL_ShowSimpleMessageBox`, y se continuaría en el menú) y salir del juego.
- Mientras se está jugando, si se pulsa la tecla *Esc*, el juego se detiene y se visualiza el *menú pausa*, que permite al menos reanudar la partida y volver al menú principal.
- Finalmente, cuando se acabe la partida deberá visualizarse el *menú fin*, una pantalla en la que además de informar al usuario si ha ganado o perdido, aparecerá un menú con opciones para volver al menú principal y para salir de la aplicación.

En todos los menús, será posible usar el ratón para seleccionar la opción elegida. Además, se implementará un sencillo estado auxiliar para ejecutar animaciones como la que aparece en el juego original cuando Mario pierde una vida.

## Detalles de implementación

### Estados del juego

El juego utiliza una máquina de estados para manejar las transiciones entre estados del juego. Implementa por tanto la clase `GameStateMachine` (incluida en el material de la práctica), que incluye como atributo una pila de estados (tipo `stack<shared_ptr<GameState>>`) y métodos `pushState`, `replaceState`, `popState`, `update`, `render` y `handleEvent`. La clase `Game` herederá privadamente de `GameStateMachine`, utilizará o extenderá sus métodos `update`, `render` y `handleEvent` y hará visibles los métodos de manejo de la pila con declaraciones como `using GameStateMachine::pushState` en la sección pública de `Game`. Debes implementar al menos las siguientes clases para manejar estados:

**Clase `GameState`:** es la clase raíz de la jerarquía de estados del juego y tiene al menos tres atributos: la colección de objetos del juego (`GameList<GameObject>`), los manejadores de eventos (`List<EventHandler*>`, véase más adelante) y el puntero al juego. Implementa los métodos `update`, `render` y `handleEvent`, y también `addEventListener` y `addObject` para añadir oyentes y objetos al estado.

**Clase `PlayState`:** implementa el juego propiamente dicho, así que incluye gran parte de los atributos y funcionalidad que antes teníamos en la clase `Game`. Los objetos de la escena se comunicarán con este estado como antes se comunicaban con `Game` (`checkCollisions`, `getMapOffset`, etc.). Además de la lista de `GameObject` heredada de `GameState`, este estado guardará una lista adicional con todos los objetos de la escena (`GameList<SceneObject>`) para calcular las colisiones.

**Clases `MainMenuState`, `PauseState` y `EndState`:** implementan respectivamente los estados del juego correspondientes a los menús *principal*, *pausa* y *fin* como subclases de `GameState`. El escenario de cada menú estará compuesto por objetos de tipo `Button` (véase más abajo) e imágenes estáticas. En el material de la práctica se proporcionan texturas para los botones y las imágenes de fondo.

**Clase AnimationState:** implementa un estado auxiliar para animaciones (véase la sección *Animaciones* más adelante) como subclase de `GameState`. Contiene como atributos un puntero al estado de juego sobre el que se aplica y el *callback* de la animación de tipo `std::function<bool()>`.

Observa que ahora la clase `Game` queda solo con los siguientes atributos básicos: los punteros a `SDL_Window` y `SDL_Renderer`, el array de texturas y la máquina de estados. La aplicación terminará cuando la pila de estados quede vacía o alternativamente utilizando un booleano `exit`. De hecho, esta clase podría pasar a llamarse `SDLApplication` pues ya no tiene nada referente al juego propiamente dicho. Los objetos de tipo `GameObject` ahora guardarán un puntero al `GameState` del que forman parte en lugar de al `Game`. Además, los objetos de tipo `SceneObject` guardarán un puntero a su `PlayState` (ese `PlayState` es el mismo `GameState` al que pertenecen, pero C++ no permite refinar el tipo de un atributo, por lo que es necesario usar dos atributos o `static_cast`).

## Botones y eventos

Los menús del juego permiten elegir entre diversas opciones mediante botones, en los que el jugador puede hacer click. Estos botones se manejan como objetos del juego, es decir, saben dibujarse, actualizarse (si es necesario), reaccionan a eventos de la SDL y emiten sus propios eventos.

**Clase Button:** por lo dicho en el párrafo anterior, es subclase de `GameObject` y `EventHandler`, con atributos para su textura y para la función o funciones a ejecutar en caso de ser pulsado (de tipo `Button::Callback`, un alias de `std::function<void(void)>`), que se invocarán desde el método `handleEvent`. Los *callbacks* se registrarán mediante un método público `connect` de la clase. Recuerda que se puede crear un objeto función para invocar al método de un objeto con cualquiera de

```
button.connect([this]() { método(); });           // expresión Lambda
button.connect(std::bind(&Clase::método, this));   // puntero al método + objeto
```

Mientras el ratón esté situado sobre el botón aparecerá a su izquierda el dibujo de un champiñón como indicación de que puede ser pulsado.

**Clase EventHandler:** se trata de una clase abstracta con un único método virtual puro `handleEvent` que recibe un `SDL_Event` (en la terminología de otros lenguajes, esto sería una interfaz). La clase `Game` se encargará como hasta ahora de capturar los eventos con `SDL_PollEvent`, pero en esta práctica los retransmitirá a todos los oyentes registrados de tipo `EventHandler`, para lo que tendrá que guardar una lista de punteros como se ha visto en clase. Las clases que capturan eventos de la SDL, `Player` y `Button`, implementarán esta interfaz y establecerán en la definición del método cómo responder a los eventos.

## Animaciones

Vamos a implementar la animación que aparece en el juego original cuando Mario pierde una vida (cambia su apariencia, pega un pequeño salto y cae sin topar con los obstáculos). El estado de juego `AnimationState` es una solución genérica para implementar animaciones programadas sobre una escena. En su constructor, recibe como argumento un estado de juego `GameState`, que se renderizará de fondo pero no se actualizará mientras esté activo el `AnimationState`. También recibe un objeto función que ejecutará en cada fotograma hasta que devuelva falso, cuando se desapilará el estado animación. La animación está definida por esta función, que típicamente moverá algún objeto de la escena.

En este caso la animación es la de la caída libre de Mario cuando es herido. En su método `hit`, cuando detecte la colisión con el enemigo, creará un `AnimationState` y le pasará un objeto función que mueva paso a paso la posición de Mario hasta que desaparezca por el borde inferior de la pantalla y entonces ejecute el código que atendía esta situación en las prácticas anteriores. El objeto función se puede implementar con una función lambda (que debería capturar `this` para poder modificar los atributos de Mario) o con una clase que implemente el operador `bool operator()`.

## Listas de objetos del juego y de la escena

En la práctica anterior había una única lista de objetos del juego de tipo `GameList<SceneObject>` como atributo de `Game`. Sin embargo, ahora cada estado del juego tendrá su propia lista de objetos de



## Funcionalidades opcionales (1 punto adicional máximo)

1. Utiliza el paquete `TTF` de SDL para manejar los distintos textos del juego (contador, botones, etc.). Es recomendable encapsular la interacción con la biblioteca en la clase `Font` que aparece en las diapositivas del tema 7 y definir un objeto del juego `Label` o semejante para colocar piezas de texto en la pantalla (`Button` podría ser subtipo de `Label`).
2. Implementa el soporte para que se registren de manera ordenada las puntuaciones de las partidas acabadas junto con un identificador del jugador. Utiliza para ello un árbol de búsqueda (tipo `std::map`) indexado por puntuación y número de orden. La clasificación debe cargarse y guardarse en un archivo. Se mostrará y se preguntará por el identificador del usuario al finalizar una partida, bien a través de la consola o en la ventana de la SDL.
3. En la pantalla de inicio, haz que la opción de iniciar la partida desde el nivel 2 solo esté activa una vez se haya superado el nivel 1. El desbloqueo puede durar hasta que el juego se cierre (más fácil) o persistir entre ejecuciones del programa (creando un archivo auxiliar).
4. En la pantalla de inicio, haz que las opciones del menú se creen dinámicamente en función de los archivos `world` disponibles en `assets/maps`. Utiliza la función `directory_iterator` para recuperar la lista de archivos sin necesidad de abrirlos.

## Entrega

En la tarea *Entrega de la práctica 3* 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* como en entregas anteriores.