Práctica 3: Frogger 3.0

Curso 2025-2026. Tecnología de la Programación de Videojuegos 1. UCM

Fecha de entrega: 2 de diciembre de 2025 a las 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 Frogger 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 con cualquiera de los mapas que se encuentren en la carpeta `assets/maps` y sobre los que se podrá iterar con las teclas de movimiento lateral. También ofrece la opción de 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, reiniciarla 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 las opciones elegidas.

Detalles de implementación

Estados del juego

El juego incorpora ahora una máquina de estados para manejar las transiciones entre estados del juego. Se utiliza 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` debe heredar privadamente de `GameStateMachine`, utilizar o extender sus métodos `update`, `render` y `handleEvent` y hacer visibles los métodos de manejo de la pila con declaraciones como `using GameStateMachine::pushState` en la sección pública de `Game`. Además, 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 (`list<GameObject*>`), los manejadores de eventos (`list<EventHandler*>`, véase más adelante), los *callbacks* pendientes (`list<DelayedCallback>`, 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` (`checkCollision`, etc.). Además de la lista de `GameObject` heredada de `GameState`, este estado guardará una lista adicional con todos los objetos de la escena (`list<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.

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 los heredados de 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 Label: representa una etiqueta de texto ubicada en una posición determinada de la ventana. Es subclase de `GameObject` y mantiene al menos como atributos su posición y su textura. La plantilla incluye texturas para todas los etiquetas necesarios, aunque si se implementa la funcionalidad opcional de la biblioteca SDL_TTF, el constructor de esta clase recibirá mejor una cadena con el texto a mostrar.

Clase Button: es subclase de `Label` y de `EventHandler`, con un atributo adicional para la función o funciones a ejecutar en caso de ser pulsado (de tipo `Button::Callback`, un alias de `std::function<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
```

El botón se resaltará de alguna forma visible cuando el ratón esté situado sobre él.

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 delegará el evento en el estado de juego activo, que a su vez lo retransmitirá a todos los oyentes de tipo `EventHandler` registrados en su lista de oyentes. Las clases que capturan eventos de la SDL, `Frog` y `Button`, implementarán esta interfaz.

Selección de mapa en el menú principal

El menú principal necesita detectar los mapas disponibles en la carpeta `assets/maps` para permitir permitir al jugador seleccionarlos e iniciar la partida con cualquiera de ellos. La clase `directory_iterator` de la biblioteca `filesystem` de C++ es un iterador sobre todos los archivos de un directorio. Sus elementos tienen un método `path` que devuelve la ruta completa del archivo y esta a su vez proporciona un método `stem` para obtener el nombre del archivo sin extensión. Por ejemplo, el siguiente código muestra los nombres de todos los archivos en la carpeta `maps`.

```
for (auto entry : std::filesystem::directory_iterator("maps"))
        cout << entry.path().stem().string() << endl;
```

La clase `MainMenuState` (o una clase auxiliar específica) mostrará el nombre de todos los mapas diponibles y permitirá cambiar de uno a otro usando las teclas de dirección ← y → del teclado. Al pulsar ↵ o hacer clic sobre el nombre, se iniciará el juego con el mapa mostrado. Una posible implementación consiste en utilizar tantos botones como mapas de los que solo uno permanezca activo (por tanto, pintándose, actualizándose y recibiendo eventos) en cada momento.

La última opción activa se ha de preservar al volver al menú principal y también entre ejecuciones del programa. Para esto último, se cargará y guardará un archivo `config.txt` con el nombre del último mapa seleccionado. Si este archivo no existe o el mapa indicado ya no está disponible, la opción seleccionada será una cualquiera. El archivo solo se guardará al término del programa.

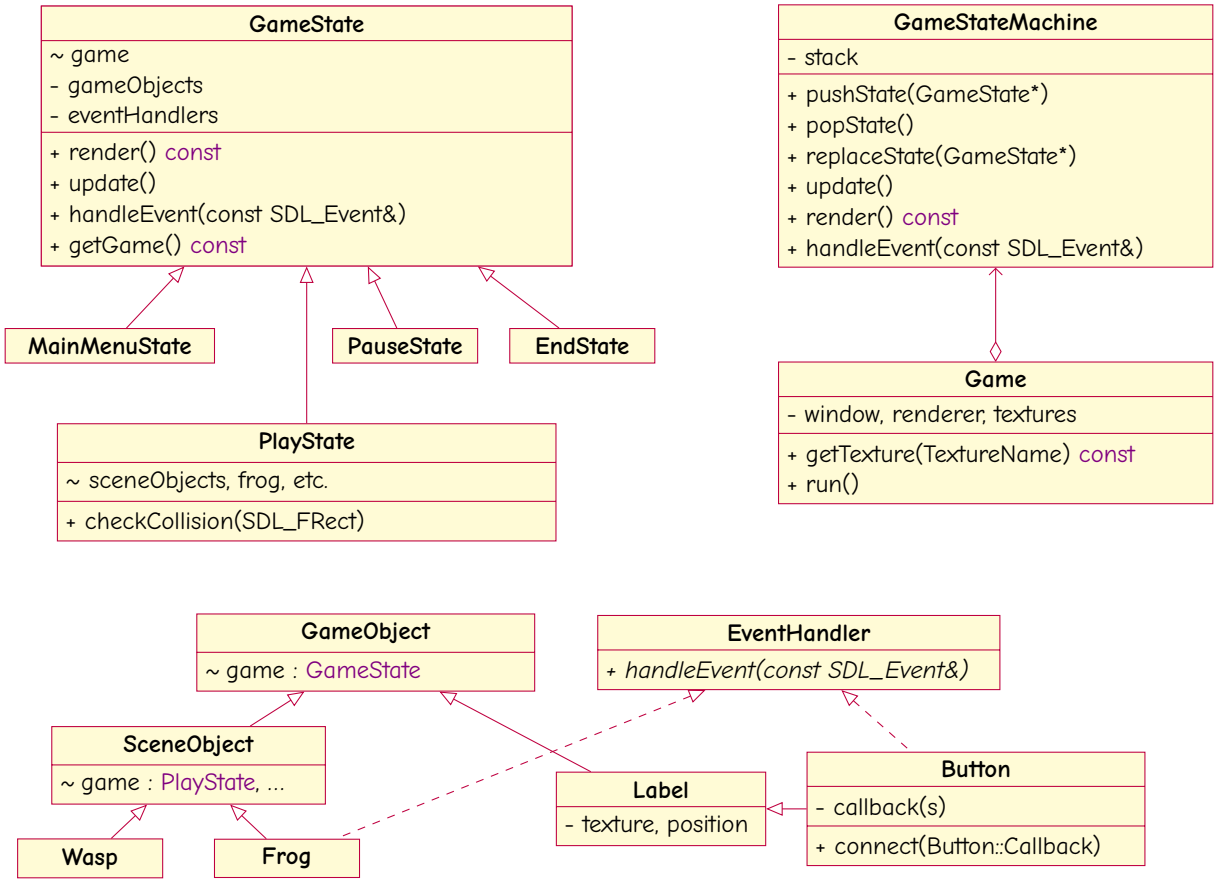

Figura 1: Diagrama incompleto de las nuevas clases del juego.

Listas de objetos del juego y de la escena

En la práctica anterior había una única lista de objetos del juego de tipo `list<SceneObject*>` como atributo de `Game`. Sin embargo, ahora cada estado del juego tendrá su propia lista de objetos de tipo `list<GameObject*>`, y el estado `PlayState` guardará también una lista `list<SceneObject*>` adicional. Esta segunda lista (cuyos elementos serán un subconjunto de los elementos de la primera) solo se usará para llamar a los métodos exclusivos de `SceneObject` (a saber, `checkCollision`). Esta segunda lista tiene un papel secundario, pues `GameState` es propietario de los objetos en sus listas de `GameObject` y debe eliminarlos al destruirse, pero no ocurre así con `PlayState` y su lista de `SceneObject`.

La avispa, el único objeto del juego que se destruye, ahora tendrá que desapuntarse de ambas listas en su retirada. Para ello,

1. se declararán en `GameState` y `PlayState` sendos alias de tipo `Anchor` para los iteradores `list<T*>::iterator` con T en `GameObject` y `SceneObject`, respectivamente. En ambas clases, el método `addObject` devolverá un elemento de su tipo `Anchor` y otro método `removeObject` lo recibirá para borrarlo de la lista.

2. la clase `GameState` declarará un tipo `DelayedCallback` como alias de `std::function<void()>` y proporcionará un método `runLater` que reciba un ejemplar de ese tipo para ejecutarlo al final de la iteración. Sería sencillo generalizar este método para recibir un tiempo de espera y ejecutar tareas con retardo arbitrario.

3. la clase `Wasp` guardará sendos iteradores para `GameState` y `PlayState`, que se fijarán cuando se cree la avispa. Cuando la avispa haya de desaparecer, se registrará un *callback* con el mecanismo anterior que libere la memoria del objeto y lo desvincule de sus listas de forma segura.

Cambios en el formato de los mapas

Esta práctica se añade la posibilidad de leer avispas desde el mapa inicial. El formato es:

`W` x y v_x v_y t

donde (x, y) es la posición inicial de la avispa, (v_x, v_y) es su velocidad en píxeles por segundo y t es su tiempo de vida restante en milisegundos. Si las avispas se implementaron como se indicó en la práctica 1, no será necesario modificar otros aspectos de su implementación.

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, y no se abuse de castings ni de consultas de tipos en ejecución.
- Asegúrate de que el programa no deje basura revisando la salida de error o la consola de Visual Studio al finalizar el programa. Cuando se compila en modo *Debug*, allí aparecerán todas las fugas de memoria detectadas. Ejecuta la práctica también con `Sanitize` para detectar errores de memoria.
- Todos los atributos deben ser privados o protegidos 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 (1 punto adicional máximo)

Además de las siguientes, se podrán contabilizar en esta práctica aquellas funcionalidades opcionales de la práctica anterior que no hayan tenido efecto sobre su nota.

1. Utiliza el paquete `TTF` de SDL para manejar los distintos textos del juego (botones, etc.). Es recomendable encapsular la interacción con la biblioteca en la clase `Font` que aparece en las diapositivas del tema 7.
2. Añade al selector de mapas del menú principal unas flechas laterales accionables con el ratón que permitan recorrer la secuencia de mapas disponibles.

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 hacer entrega de la práctica como indican las instrucciones que se proporcionan 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.