Práctica 2: Frogger 2.0

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

Fecha de entrega: 18 de noviembre de 2025 a las 12:00

El objetivo principal 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 las modificaciones que se explican a continuación. En cuanto a funcionalidad, el juego presenta las siguientes novedades:

1. Se sustituyen los troncos de la primera y la cuarta fila del río por grupos de tortugas que remontan la corriente, tal como ocurre en el juego original. Uno de los grupos de cada fila además se sumerge periódicamente. Esto se corresponde con dos de los apartados opcionales de la práctica 1.
2. El jugador ahora tiene la opción de reiniciar la partida en cualquier momento del juego.

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 necesarios para mejorar la claridad y reusabilidad del código. Como norma general, cada clase se corresponderá con un par de archivos `.h` y `.cpp`. Las nuevas clases de esta práctica son las siguientes:

Clase GameObject: esta clase abstracta 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 una destructora virtual, un atributo con un puntero al juego y un constructor protegido que reciba ese puntero.

Clase SceneObject: subclase de `GameObject` de la que descienden todos los elementos de la escena (es decir, todas las clases obligatorias de la práctica anterior). Sus atributos mantienen la posición del objeto en la ventana, sus dimensiones y su textura. En consecuencia, hay información suficiente para definir un método `render` por defecto, aunque algunas subclases necesiten sobrescribirlo. Además, declara un método público virtual `checkCollision` con el mismo significado que en la práctica anterior, y un método protegido `getBoundingBox` que devuelva la caja de corte del objeto.

Clase Crosser: subclase de `SceneObject` que engloba a todos los objetos que cruzan la escena de un lado a otro, a saber, las plataformas y los vehículos. Su método `update` implementa este comportamiento utilizando como atributos un vector velocidad y el desplazamiento horizontal que se le ha de aplicar para reaparecer por el lado opuesto (véase la ilustración de la práctica anterior).

Clase Platform: subclase de `Crosser` que aúna a todas las plataformas sobre la que se puede apoyar la rana. La clase `Log` y la nueva clase `TurtleGroup` heredarán de `Platform`. No declara atributos nuevos, copia el constructor de su clase madre con **`using`** y define simplemente el método `checkCollision` con la lógica de una plataforma.

Clase TurtleGroup: representa un grupo de tortugas que remonta el río junto a los troncos. Es una plataforma (subclase de `Platform`) y se construye tomando como argumentos, entre otros, el número de tortugas del grupo y un booleano que indique si el grupo se sumerge o no. Se añadirán los atributos y se sobrescribirán los métodos que hagan falta para implementar su comportamiento.

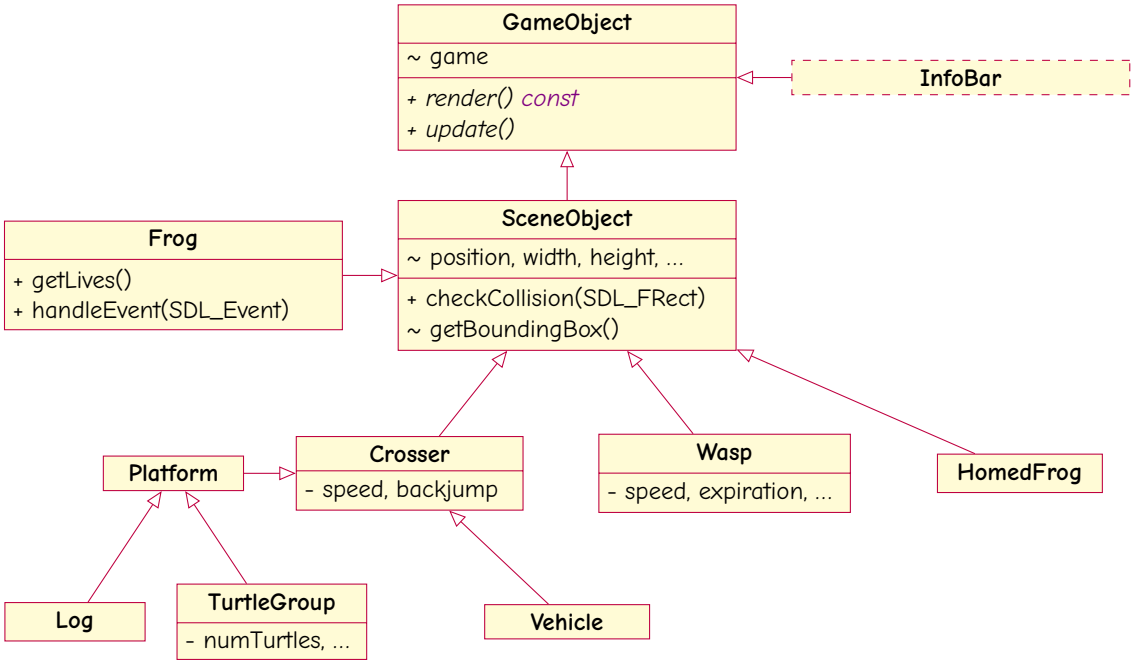


Figura 1: Digrama de clases principales del juego.

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

Los objetos del juego se han mantenido en la primera práctica como punteros o vectores de punteros para cada tipo de objeto. El polimorfismo que ahora conocemos permite simplificar esto enormemente, pues un solo contenedor basta para almacenar todos los objetos a través de punteros a su 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 `checkCollision` declarados en la clase general. En consecuencia, utilizaremos un atributo de tipo `std::list<SceneObject*>` para almacenar y manipular los objetos de la escena.

Eliminación de las avispas y otros objetos dinámicos

Es habitual que en los juegos se creen y se destruyan objetos dinámicamente, si bien en esta práctica solo ocurre con la avispa. Cuando `Game` crea una avispa nueva, la añade a la lista de objetos del juego sin ninguna complicación. En cambio, la eliminación es algo más complicada, pues quien detecta que la avispa ha muerto es la propia avispa y no basta con que destruya su objeto, tiene que avisar al juego para que la elimine de la lista polimórfica. Una opción sería extender el método `isAlive` de la primera práctica a `SceneObject` y dar una pasada de eliminación al final de la actualización. Otra opción, la que seguiremos, es que las avispas guardan como atributo su propio iterador en la lista polimórfica de objetos. Cuando la avispa muera, llamará a un método público `deleteAfter` de `Game` con su iterador, que el juego almacenará temporalmente y luego usará para borrar los objetos retirados al final de la actualización. Por comodidad, define un alias

```
using Anchor = std::list<SceneObject*>::iterator;
```

para el tipo del iterador en `Game`.

Reiniciar la partida

En esta práctica añadiremos la posibilidad de reiniciar en cualquier momento la partida a su configuración inicial (volviéndola a cargar del archivo). Cuando el usuario active esta opción, se eliminarán todos los objetos del juego y se volverán a crear en sus posiciones y estados iniciales. No está permitido volver a ejecutar el programa o destruir la ventana de la SDL. El reinicio se activa al pulsar la tecla 0 (cero), evitando reinicios accidentales. Para ello se puede elegir entre una de las siguientes medidas de precaución:

1. Que las teclas Ctrl y ⇧ deban estar pulsadas cuando se pulsa la tecla 0.
2. Que aparezca un cuadro de dialogo de confirmación usando `SDL_ShowMessageBox`.

Indica en el archivo `info.txt` qué opción se ha escogido.

Colisiones entre objetos

Las colisiones entre objetos funcionan igual que en la práctica anterior, salvo que se usa el método virtual `checkCollision` sobre el array polimórfico de objetos `SceneObject`.

Cambios en el formato de los mapas

Esta práctica extiende el formato de los mapas con la representación de los grupos de tortugas:

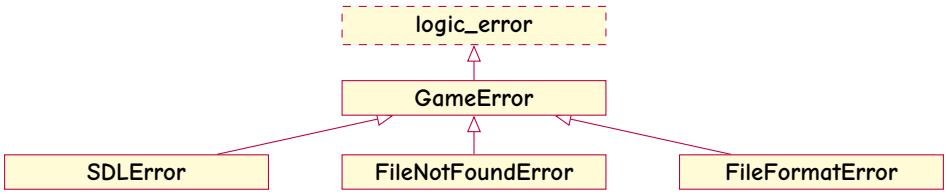
`T` x y v_x n h

donde (x, y) es la posición del grupo de tortugas (esquina superior izquierda de la primera tortuga), v_x es su velocidad horizontal, n es el número de tortugas en el grupo y h es un booleano (`0` o `1`) que indica si las tortugas se hunden o no. El archivo `turtles.txt` incluye varios grupos de tortugas donde antes de se encontraban los troncos temporales.

La lectura se ha de implementar distribuida entre las clases y cada una de ellas debe leer sus propios atributos.

Jerarquía de excepciones

Los errores del juego se ha de manejar mediante las siguientes clases de excepciones. Las excepciones irrecuperables se deben capturar en el `main` y la función `SDL_ShowSimpleMessageBox` puede utilizarse para mostrar el mensaje de error al jugador.



GameError: hereda de `std::logic_error` y sirve como superclase de todas las demás excepciones que definiremos. No proporciona nueva funcionalidad y simplemente reutiliza el constructor y método `what` de `logic_error`. Sin embargo, su tipo permite filtrar las excepciones propias del juego en bloques **`catch`**.

SDLError: hereda de `GameError` 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.

FileNotFoundError: hereda de `GameError` 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.

FileFormatError: hereda de `GameError` 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 revisando la salida de error o la consola de Visual Studio al finalizar el programa. Cuando se compila en modo *Debug*, allí aparecerá un volcado de todos las fugas de memoria detectadas.
- 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. Limita el tiempo máximo que puede tardar el jugador en llegar a una casa, como en el juego original. Se dispondrá inicialmente de 60 segundos (u otro valor conveniente) y el tiempo se restablecerá al llegar a una casa o perder una vida. Si el tiempo se agota, el jugador perderá una vida y volverá a la posición incial. El tiempo restante se mostrará en la pantalla o periódicamente en la terminal.
2. Implementa o extiende la 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 el tiempo disponible (como una barra decreciente).
3. Anima el movimiento de la rana utilizando el segundo frame de `frog.png`. Fijado un valor N, la rana completará un salto en N ciclos de actualización, moviéndose una N-ésima parte del salto en cada ciclo y cambiando de frame en parte de ese proceso.
4. Añade la opción de cargar y/o guardar partida en cualquier momento durante el juego pulsando Ctrl + L o Ctrl + S. La carga de partidas puede reutilizar la carga del mapa inicial. La función de guardado se puede implementar mediante un método público virtual `save` de `SceneObject`. Se pueden usar nombres de archivo fijos, pedir el nombre en el terminal o utilizar `SDL_ShowOpenFileDialog`.
5. Incorpora sonido al salto utilizando el archivo `jump.wav` de la plantilla. Necesitarás usar la API de audio de SDL, además de pasar `SDL_INIT_AUDIO` a `SDL_Init`.

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 la práctica siguiendo las instrucciones que aparezcan en el campus virtual. La raíz de la entrega 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.