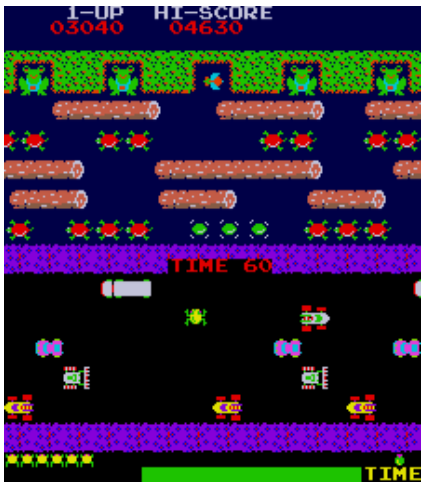
Práctica 1: Frogger 1.0

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

Fecha de entrega: 28 de octubre de 2025



En esta práctica implementaremos en C++ una versión simplificada del conocido juego Frogger, desarrollado originalmente como juego de arcade por Konami en 1981. Se utilizará la biblioteca SDL para manejar la entrada/salida del juego. Tomaremos como referencia la versión que se puede encontrar en `happyhopper.org` con algunas simplificaciones:

- No hay serpientes, nutrias, cocodrilos, ranas azules, ni tortugas en la travesía del río.
- La velocidad de movimiento de los objetos no cambia con el tiempo o con el número de intentos.
- No se controla el tiempo restante ni la puntuación.
- No se muestra el número de vidas en pantalla.

En futuras versiones de la práctica o en los apartados opcionales se recuperarán 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 necesarios 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 (incluida en la plantilla): 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`. Además de la constructora, implementa métodos para consultar las componentes `x` e `y`, así como operadores para la suma, resta, producto escalar de vectores y producto de un vector por un escalar. En el mismo módulo, define con **`using`** un alias `Point2D` para el tipo, que usaremos cuando nos refiramos a un punto en lugar de una dirección.

Clase Vehicle: representa un vehículo que circula por la carretera de la mitad inferior de la escena. Contiene un puntero a su textura, una posición (tipo `Point2D`) y una velocidad (tipo `Vector2D`). Implementa un constructor y métodos para dibujarse (**`void`** `render()` **`const`**), actualizarse (**`void`** `update()`) y detectar colisiones (`Collision checkCollision(`**`const`** `SDL_FRect&)`). Todos los vehículos ( ,  ,  ,  y ) son instancias de esta clase, cada uno con su textura y velocidad correspondientes.

Clase Log (): representa un tronco que navega por el río de la mitad superior de la escena. Se compone de los mismos atributos y métodos que se han enumerado para `Vehicle`.

Clase Wasp (🐝): representa una avispa que se mueve o queda parada en el tablero. Además de los atributos y métodos de las clases anteriores, contiene un atributo para su tiempo de vida en milisegundos (o su fecha de caducidad, si se prefiere) y un método `bool isAlive() const` para comprobar si este tiempo ha transcurrido. Cuando eso ocurra la avispa desaparecerá como se explica más adelante.

Clase HomedFrog (🐸): representa una rana que ha llegado al nido. Mantiene los atributos e implementa los métodos habituales, aunque no tiene velocidad al ser un objeto estático.

Clase Frog (🐸): es la rana controlada por el jugador humano. Sus atributos son al menos un puntero al juego, un puntero a su textura, su posición actual, la última dirección de movimiento y el número de vidas que le quedan (inicialmente 3). Implementa también métodos para construirse, dibujarse (`render`), actualizarse, es decir, moverse (`update`), y manejar eventos del teclado (método `void handleEvent(const SDL_Event&)`), que determinan el estado de movimiento.

La rana perderá una vida y volverá a la posición inicial cuando sea atropellada por un vehículo, caiga al río, choque con una avispa, caiga sobre un nido ocupado o una posición que no sea nido más arriba del río o sobrepase el borde de la escena a bordo de un tronco. También volverá a la posición inicial, sin perder una vida, cuando llegue a un nido. En ese caso dejará además una `HomedFrog` en el sitio. Cuando la rana pierda todas sus vidas, el juego se cerrará.

Clase Game: contiene, al menos, punteros a la ventana y al renderizador, a los objetos 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 (`render`), actualizarlo (`update`) y manejar eventos (método `void handleEvents()`). Además, sin abusar, tendrá otros métodos auxiliares que hagan falta como `checkCollision` (ver más abajo).

Carga de texturas

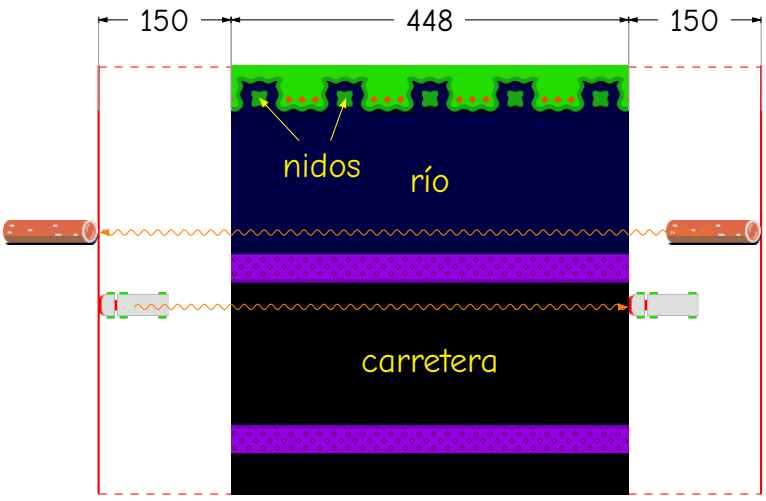
Las texturas con las imágenes necesarias deben cargarse durante la inicialización del juego y guardarse en un array estático de tipo `array<Texture*, NUM_TEXTURES>` que aparece en la plantilla. El método `Texture* getTexture(TextureName)` permitirá acceder a las texturas cuando hayan de ser usadas. Sigue las instrucciones de la plantilla para especificar las texturas y cargarlas.

Renderizado del juego

El bucle principal del juego (método `Game::run`) invoca al método `Game::render`, que borra la pantalla (con `SDL_RenderClear`), delega el renderizado a los diferentes objetos del juego llamando a sus respectivos métodos `render` y finalmente presenta la escena en la pantalla (llamando a la función `SDL_RenderPresent`). Cada objeto del juego sabe cómo pintarse, pues 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.

Movimiento de los objetos

El bucle principal del juego (método `Game::run`) desencadena periódicamente actualizaciones de su estado llamando al método `Game::update`. Este a su vez llama sucesivamente 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 vehículos, troncos y avispas se mueven cada uno según su vector velocidad. Los vehículos y troncos además desaparecen por un extremo de la escena para volver a aparecer por el extremo opuesto. Para facilitar esta operación, conviene prolongar el movimiento de los objetos entre bastidores, en una extensión oculta de la escena. Así, cuando un objeto choque contra un borde oculto de la escena (coloreado en rojo en el dibujo siguiente), será trasladado al borde visible opuesto.



Por otro lado, el movimiento de la rana está controlado por el usuario a través del teclado. Cada pulsación de cualquiera de las flechas de dirección (u otras teclas equivalentes) supone el desplazamiento de una casilla en dicha dirección de la escena. 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 `Game::handleEvents`) delega el manejo de este evento en el objeto del jugador (llamando a su método `handleEvent`), que actualizará su dirección de movimiento en consecuencia (izquierda, derecha, arriba, abajo o quieto). La 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 se desplazará a trompicones.

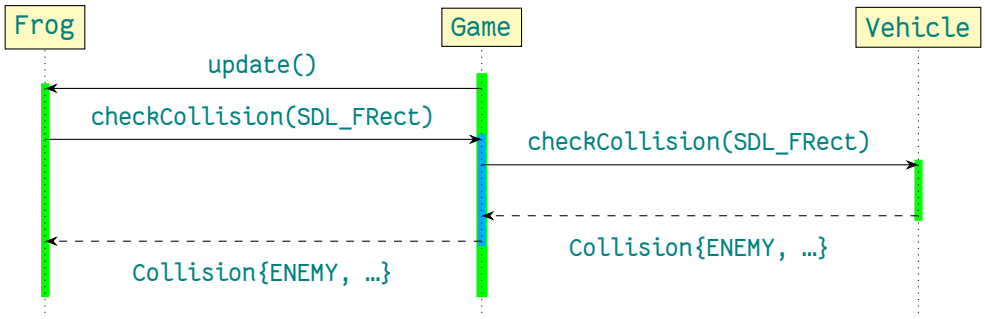
Manejo de colisiones

Los objetos que puedan colisionar con la rana tendrán un método

```
Collision checkCollision(const SDL_FRect&);
```

donde `Collision` es un tipo **`struct`** con un atributo de tipo enumerado `Type` (con valores `NONE`, `ENEMY` y `PLATFORM`) y un vector velocidad de tipo `Vector2D`. Estos métodos reciben un rectángulo que se intersecará con la caja delimitadora del objeto (que se puede calcular a partir de su posición, altura y anchura) usando la función `SDL_HasRectIntersectionFloat`. Si la intersección es no vacía, los vehículos, avispas y ranas en casa devolverán una colisión de tipo `ENEMY`, mientras que los troncos devolverán una colisión de tipo `PLATFORM` con la velocidad del tronco. Eso permitirá a la rana desplazarse a la misma velocidad del tronco sobre el que está subida.

La rana llamará desde su método `update` con su caja delimitadora a un método `checkCollision` de `Game` para comprobar si hay algún obstáculo en su posición actual. Este método a su vez llamará a los métodos `checkCollision` de todos los objetos de la escena que dispongan de él hasta encontrar uno donde la respuesta sea afirmativa. Se asume que no hay colisiones con más de un objeto.



Eliminación de los objetos del juego

La mayoría de objetos del juego permanecen activos durante toda la ejecución y solo han de ser eliminados en el destructor de `Game`. La excepción son las avispas, que se crean y destruyen durante la

partida. Al finalizar la etapa de actualización, el método `update` de `Game` deberá revisar la colección de objeto de tipo `Wasp` eliminando aquellos cuyo método `isAlive` devuelva `false`.

Generación aleatoria de avispas

Las avispas (clase `Wasp`) han de aparecer aleatoriamente durante el juego y permanecer durante un tiempo también aleatorio en alguna de los nidos de la rana. Esto requiere generar números aleatorios utilizando la cabecera `random` de la biblioteca estándar de C++ (se darán más explicaciones en clase). La clase `Game` debe mantener como atributo un generador de números pseudoaleatorios (por ejemplo, un objeto de la clase `mt19937_64`) e implementar un método público

```
int Game::getRandomRange(int min, int max) {
        return uniform_int_distribution<int>(min, max)(randomGenerator);
}
```

que devuelva un número en el rango `min`-`max` con probabilidad uniforme. Si se quieren obtener diferentes números aleatorios en diferentes ejecuciones del programa, se debe establecer la *semilla* del generador de números aleatorios (el argumento de su constructora) con un valor distinto cada vez. Por ejemplo, se puede utilizar `time(nullptr)` (el segundo actual) o `random_device()` (ruido ambiental).

Formato de ficheros de configuraciones iniciales

Las posiciones iniciales de la rana, los vehículos, los troncos y el resto de elementos del juego se podría fijar en el mismo código, pero es más versátil cargar esta información desde un archivo de texto. Los archivos que el programa deberá reconocer introducen en cada línea un objeto distinto. La línea comienza con un identificador del tipo de objeto (`F` para la rana, `V` para un vehículo, `L` para un tronco y `T` para una tortuga), seguido de su posición y finalmente de los atributos extra de cada personaje, separados por espacios. La posición será un par de números en coma flotante y miden el desplazamiento en píxeles de la esquina superior izquierda del objeto respecto a la esquina superior izquierda del mapa. Los objetos que tiene atributos extra son: la rana, con el número de vidas; el vehículo, con su velocidad horizontal (un número entero con signo, en píxeles por segundo) y el tipo de vehículo (un número del 1 al 4); y el tronco, también con su velocidad horizontal. Las líneas que comiencen por almohadilla (`#`) se ignorarán utilizando el método `ignore` del flujo de archivo.

Cada objeto del juego dispondrá de un constructor que reciba un argumento de tipo `std::istream&` del que leer sus propios datos (salvo el identificador, que se leerá en un método `loadMap` de `Game` para llamar al constructor adecuado). El archivo se abrirá como un objeto `std::ifstream` en el `Game` con la ruta `../assets/maps/default.txt`, declarada como una constante.

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, error en el formato del fichero de mapa y error de SDL. Puesto que son todos ellos errores irrecuperables, 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 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.

- Asegúrate de que el programa hace un correcto uso de la memoria (se inicializan todas las variables y atributos antes de leerlos, no se libera dos veces un mismo bloque de memoria, etc.). El modo de compilación *Sanitize* activa un comprobador de memoria que puede detectar estos errores.
- Todos los atributos deben ser privados excepto quizás algunas constantes del juego definidas como atributos estáticos.
- Define las constantes que sean necesarias, mejor con **`constexpr`** que con **`const`**. 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. Usa preferiblemente nombres en inglés.

Funcionalidades opcionales

1. 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.
2. Implementar los grupos de dos o tres tortugas que remontan el río en el juego original. Utiliza una nueva clase `Turtles` de forma semejante al resto de objetos del juego. Las tortugas sustituyen a los troncos de la primera y cuarta fila del río (contando desde la carretera).
3. Haz que las tortugas del apartado anterior se sumerjan y emerjan periódicamente como en el juego original. Cuando las tortugas estén sumergidas no se comportarán como un apoyo para evitar caer al río.

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. 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.

Entrega intermedia el 21 de octubre: un 20% de la nota se obtendrá el día 21 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 la escena con los objetos en movimiento y que se pueda controlar a la rana.