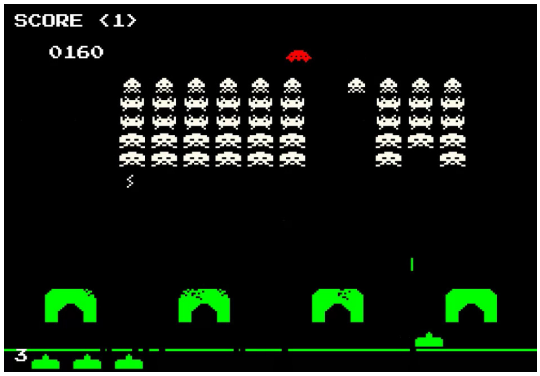
# Práctica 1: Space Invaders 1.0

## Curso 2023-2024. Tecnología de la Programación de Videojuegos 1. UCM

Fecha de entrega: 7 de noviembre de 2023

El objetivo de esta práctica es implementar en C++/SDL una versión simplificada del conocido juego *Space Invaders* creado por Toshihiro Nishikado en 1978 (véase es.wikipedia.org/wiki/Space Invaders). Se utilizará la biblioteca SDL para manejar la entrada/salida del juego. Tomaremos como base para la mecánica del juego la versión del Space Invaders disponible para jugar online en Minijuegos. A continuación se detallan las principales simplificaciones y diferencias que nuestro juego tendrá respecto al original indicado:




- El juego acaba cuando se han derribado todas las naves alienígenas (el jugador gana) o bien cuando el cañón láser se queda sin vidas (el jugador pierde). No hay por tanto concepto de puntuación, con las simplificaciones que ello conlleva, ni de paso de nivel. Se deja libertad respecto a la manera en la que se informa al usuario tanto del número de vidas restantes, como de si gana o pierde cuando el juego acaba (en consola, en ventana emergente o en la propia ventana SDL).
- La velocidad de los alienígenas es siempre la misma y no descienden de su posición inicial. No hay animaciones de los alienígenas ni de su explosión.
- Los alienígenas no tienen inteligencia alguna y disparan aleatoriamente sin tener en cuenta la posición del cañón o de los búnkeres. No se sigue la restricción del juego original de que no haya más de un disparo del jugador o más de un disparo alienígena en la escena en cada momento.
- No consideramos el platillo volante rojo que cruza la pantalla cada cierto tiempo en el juego original.

## 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 necesario para mejorar la claridad y reusabilidad del código. Cada clase se corresponderá con un par de archivos .h y .cpp.

**Clase Texture** (disponible en el CV): permite manejar texturas SDL. Contiene un puntero a la textura SDL e información sobre su tamaño total y los tamaños de sus frames. 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:** representa vectores o puntos en dos dimensiones y por tanto incluye dos atributos (`x` e `y`) de un tipo genérico `T` (usando plantillas). Implementa, además de la constructora, métodos para consultar las componentes `x` e `y`, y para la suma, resta, producto escalar de vectores y producto de un vector por un escalar. La suma, resta y productos deben implementarse como operadores. En el mismo módulo define un alias `Point2D` para el tipo.

**Clase Bunker:** contiene la posición del búnker (tipo `Point2D`), el número de vidas restantes y un puntero a su textura. Implementa un constructor, métodos para dibujarse (método `render`), para actualizarse (método

`update`) y para recibir daño (método `hit`). Cada vez que reciba daño el número de vidas ha de disminuir y la apariencia se ha de ajustar en consecuencia. El método `update` devolverá `false` si el búnker ha perdido todas sus vidas y eso provocará que el juego lo elimine.

**Clase Alien:** contiene al menos la posición actual del alienígena (tipo `Point2D`), el índice de su subtipo (de tipo `int`), un puntero a su textura y un puntero al juego. Este último es necesario para que el alienígena conozca la dirección de movimiento de sus semejantes y pueda informar de que ya no se puede desplazar más en ella (ver más abajo cómo se maneja el desplazamiento). Implementa además métodos para construirse, dibujarse (método `render`), actualizarse (método `update`), y recibir daño. En la actualización, además de moverse, el alienígena podrá disparar aleatoriamente un láser una vez trascurrido el periodo de recarga de su disparo anterior. El método `update` devolverá `false` si el alienígena ha sido alcanzado por un láser del cañón.

**Clase Cannon:** al igual que los anteriores, sus atributos incluyen su posición actual, un puntero a su textura y un puntero al juego (que utilizará para lanzar láseres). Además, mantiene la dirección actual del movimiento, el número de vidas que le quedan y el tiempo restante de recarga del láser. Implementa también métodos para construirse, dibujarse (`render`), actualizarse, es decir, moverse (método `update`), recibir daño (método `hit`) y manejar eventos del teclado (método `handleEvent`), que determinan el estado de movimiento y permiten lanzar el láser (barra espaciadora).

**Clase Laser:** contiene su posición (tipo `Point2D`), una velocidad (tipo `Vector2D`) y un booleano que indica si procede de un alienígena o de la nave que controla el jugador. Implementa métodos para construirse, dibujarse (`render`) y actualizarse (método `update`). La actualización consiste en avanzar de acuerdo a su velocidad y comprobar si ha acertado a algún objetivo. Los láseres lanzados por el jugador hieren a los alienígenas, mientras que los lanzados por los alienígenas dañan al jugador. Los láseres hieren a los búnkeres independientemente de su origen. Cuando dos disparos de origen distinto se cruzan se anulan mutuamente. Se pueden dibujar como un rectángulo de color con `SDL_RenderFillRect`.

**Clase Game:** contiene, al menos, el tamaño de la ventana, punteros a la ventana y al *renderer*, a los elementos del juego (usa 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 (método `render`), actualizar (método `update`), manejar eventos (método `handleEvents`), obtener la dirección de movimiento de los alienígenas (método `getDirection`), informar de que no es posible moverse otra iteración más en la dirección actual (método `cannotMove`) y disparar láseres (método `fireLaser`).

## Carga de texturas

Las texturas con las imágenes del juego deben cargarse durante la inicialización del juego (en la constructora de `Game`) y guardarse en un array estático (tipo `array`) de `NUM_TEXTURES` elementos de tipo `Texture*`) cuyos índices serán valores de un tipo enumerado (`TextureName`) con los nombres de las distintas texturas. Los nombres de los ficheros de imágenes y los números de frames en horizontal y vertical, deben estar definidos (mediante inicialización homogénea) en un array constante de `NUM_TEXTURES` estructuras en `Game.cpp`. Esto permite automatizar el proceso de carga de texturas. Utiliza una constante auxiliar `textureRoot` para evitar repetir la parte común de la ruta en cada entrada del array de texturas.

## Renderizado del juego

En el bucle principal del juego (método `run`) se invoca al método del juego `render`, que simplemente delega el renderizado a los diferentes objetos del juego (cañón, alienígenas, láseres y búnkeres) llamando a sus respectivos métodos `render`, y finalmente presenta la escena en la pantalla (llamada a la función `SDL_RenderPresent`). Cada objeto del juego sabe cómo pintarse, en concreto, 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, quién realizará el renderizado real (llamada a la función `SDL_RenderCopy`).

## Movimiento del cañón y de los alienígenas

Los movimientos de los personajes del juego se organizan desde la clase `Game` y se van delegando de la siguiente forma: el método `update` del juego (invocado desde el bucle principal del método `run`) va llamando 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 alienígenas se mueven coordinadamente en la misma dirección, por lo que han de consultar la dirección común de movimiento a través del método `getDirection` de `Game`. De ahí que los objetos del juego tengan un puntero al juego. Esta dirección se ha de invertir *al final de la actualización* si alguno de los alienígenas ha llamado al método `cannotMove` de `Game`. Hay que tener en cuenta que `cannotMove` no se refiere al movimiento de la iteración actual, que sí se ha podido hacer, sino al que correspondería a la iteración siguiente.

El movimiento del cañón está controlado por el usuario a través del teclado. El cañón se desplazará únicamente en horizontal y a velocidad constante mientras se mantengan pulsadas las flechas izquierda o derecha (u otras teclas equivalentes) del teclado. La implementación de esta funcionalidad se llevaría a cabo de la siguiente forma: cuando se pulsa o suelta una tecla de desplazamiento, el bucle de manejo de eventos (en el método `handleEvents`) delega el manejo de este evento en el objeto cañón (mediante invocación a su método `handleEvent`), que actualizará su dirección de movimiento en consecuencia (izquierda, derecha o quieto). El cañón (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 el cañón se desplazará a trompicones.

**Números aleatorios:** para el disparo de los alienígenas se necesitará 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 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 mapas y configuraciones iniciales

Inicialmente, supondremos que la escena contiene una cuadrícula fija de alienígenas (por ejemplo, 4x11 como en el juego original), que se puede generar con un par de bucles sencillos. Más adelante, la posición inicial de los elementos en la escena se ha de leer de un fichero de texto, donde cada línea representa un elemento. El primer número de cada línea indica el tipo de elemento (0 = cañón, 1 = alienígena y 2 = búnker), al que siguen dos números con las coordenadas de su posición desde la esquina superior izquierda (asumiendo una pantalla de 800x600 unidades). Las líneas que describen alienígenas contienen un cuarto número con el subtipo de alienígena de que se trata (0 = disparador, 1 = verde y 2 = rojo).

## 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, y error de SDL. Puesto que en principio 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. 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 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. Implementar la animación de los alienígenas. La textura proporcionada incluye dos variantes para cada tipo de alienígena que pueden alternarse secuencialmente.
2. Extender la mecánica del juego para llevar cuenta de la puntuación obtenida. En el juego original los alienígenas disparadores valen 30 puntos, los verdes 20 y los rojos 10. La puntuación actual del juego se puede mostrar por consola cuando se consigue nueva puntuación.
3. Implementa una clase `InfoBar` cuyo método `render` se encargaría de mostrar en la ventana SDL una barra de información del juego que incluya al menos el número de vidas restantes y posiblemente también la puntuación actual.
4. Implementar el aumento de velocidad de los alienígenas y/o su descenso según avanza el juego, al igual que en el original. El jugador pierde la partida si bajan de un determinado umbral.
5. Extendiendo el formato de los mapas iniciales, implementar el soporte para guardar y cargar partidas.

## 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 un fichero comprimido (.zip) que contenga la carpeta de la solución y el proyecto limpio de archivos temporales (asegúrate de borrar la carpeta oculta `.vs` y ejecuta en Visual Studio la opción «limpiar solución» antes de generar el .zip). 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 31 de octubre:** un 20% de la nota se obtendrá el día 31 de noviembre 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 construya la escena original, mueva los alienígenas y permita que el cañón se mueva.