Fundamentos de la programación

Práctica 1. Combate aéreo

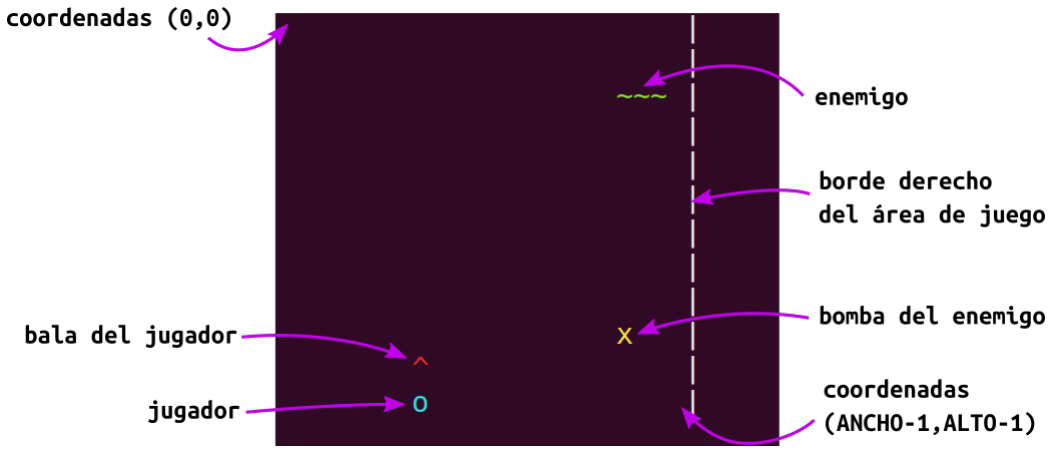
Indicaciones generales:

- La línea 1 del programa (y siguientes) deben contener los nombres de los alumnos de la forma:

  `// Nombre Apellido1 Apellido2`

- **Lee atentamente** el enunciado e implementa el programa tal como se pide, con la representación y esquema propuestos, y con los requisitos que se especifican.
- El programa, además de correcto, debe estar bien estructurado y comentado. Se valorarán la claridad, la concisión y la eficiencia.
- La entrega se realizará a través del campus virtual, subiendo únicamente el archivo `Program.cs`, con el programa completo.
- El **plazo de entrega** finaliza el 30 de noviembre.

Vamos a implementar un juego elemental de combate aéreo en la consola. El espacio de juego será una cuadrícula de dimensiones `ANCHO` × `ALTO` donde se representará el jugador y el enemigo, así como sus respectivos disparos:



El espacio de juego está determinado por la esquina superior izquierda, con coordenadas `(0,0)`, y la inferior derecha, con coordenadas `(ANCHO-1,ALTO-1)`. El jugador, representado con `O`, puede moverse por todo el área de juego con las telcas habituales `"asdw"`. También puede disparar balas hacia arriba (en línea recta) con la tecla `"l"`, representadas con ^. El enemigo, representado con ~~~, puede moverse en la *mitad superior del área de juego*, entre las coordenadas `(0,0)` y `(ANCHO-1,ALTO/2)`. Puede disparar bombas hacia abajo, representadas con `X`.

La velocidad de refresco (tiempo entre *frames*) vendrá dada por una constante `DELTA` (en milisegundos). En cada frame el jugador puede desplazarse una posición o disparar una bala si no hay ya una en pantalla (no puede haber dos balas simultáneamente). Las balas arrancarán justo desde la posición por encima del jugador y se desplazarán una posición hacia arriba en cada frame. El enemigo puede desplazarse una unidad a izquierda o derecha y también arriba o abajo (es decir, podría desplazarse una unidad en diagonal en un solo frame). En cada frame disparará una bomba si no hay ninguna cayendo (no puede haber dos bombas simultáneamente). Las bombas arrancarán justo desde la posición inferior al centro del enemigo y se desplazarán una posición hacia abajo en cada frame.

Tanto las balas como las bombas continúan su movimiento hasta salir del área de juego o bien colisionar. Si una bala colisiona con el enemigo el jugador gana la partida, mientras que si una

bomba colisiona con el jugador, el ganador será el enemigo. También es posible la colisión entre bala y bomba, en cuyo caso se anulan ambas (y desaparecen de la pantalla).

Además de las constantes `DELTA`, `ANCHO` y `ALTO`, el estado interno del juego vendrá determinado por las posiciones de las entidades implicadas:

- Las coordenadas `(jugX,jugY)` del jugador
- Las coordenadas `(eneX,eneY)` correspondientes a *la posición central* del enemigo (que en total ocupa tres posiciones en el área de juego).
- Las coordenadas `(balaX,balaY)` de la bala lanzada por el jugador, si hay bala activa en ese momento. Para denotar que no hay bala activa utilizaremos un valor especial fuera del área de juego: `balaX = -1`.
- Las coordenadas `(bombaX,bombaY)` de la bomba lanzada por el enemigo. Como antes, cuando no hay bomba cayendo tendremos un valor especial `bombaX = -1` (hasta que se lance la siguiente).

En cada vuelta del bucle principal se actualizará el estado del juego, realizando las siguientes tareas, **por este orden**:

- **Input de usuario**. Si hiciésemos la lectura de input con el habitual `Console.ReadLine()`, la entrada sería *bloqueante*, es decir, el programa pararía hasta recibir entrada de teclado y la tecla *intro*. Para evitar esta parada y que el juego *fluya* utilizaremos una lectura no bloqueante: si hay pulsación de teclado se recoge el carácter correspondiente; si no, continúa la ejecución. Esto puede hacerse del siguiente modo:

```
if (Console.KeyAvailable) { // si hay pulsación
  string s = (Console.ReadKey(true)).KeyChar.ToString(); // recogemos tecla
  ...
```

  De este modo si no hay pulsación el flujo de programa continúa sin detenerse y si hay pulsación, tendremos el valor correspondiente en la cadena `s`. De acuerdo con este valor, después el jugador se desplazará una posición, lanzará una bala, o no hará nada (si no hay pulsación).

- **Movimientos**. En cada frame hay varias actualizaciones del estado del juego:

  - El enemigo podrá desplazarse aleatoriamente una unidad en cualquier dirección, tal como se ha explicado arriba. Para generar números aleatorios se puede inicializar el generador al principio del programa con (**el generador debe inicializarse una sola vez en todo el programa**):

```
Random rnd = new Random();
```

    Después, cada vez que necesitemos un número aleatorio haremos:

```
int aleatorio = rnd.Next (i, j);
```

    Importante: devuelve un entero aleatorio del intervalo *[i,j-1]*, i.e., *j* queda excluido.

    Deberemos controlar que las 3 coordenadas correspondientes al enemigo caigan dentro de los límites del área de juego. Si no fuese así, no aplicaríamos el desplazamiento. Por ejemplo, si está pegado al borde derecho y (aleatoriamente) debe ir a la derecha, no se aplica el movimiento y queda en la misma posición en ese frame.

  - Si hay bomba en el área de juego, se desplazará una unidad hacia abajo. Si se sale del área de juego se hará `bombaX = -1` para eliminarla. Si no hay bomba (en el siguiente frame) se generará una nueva, tal como se ha explicado.

- Si hay bala activa, se desplazará una unidad hacia arriba. Si se sale del área de juego, se hará `balaX = -1` para eliminarla. Cuando el jugador dispare, se pondrán las coordenadas correspondientes a la nueva bala.

- **Control de colisiones**. Una vez determinada la nueva situación de todas las entidades del juego, hay que determinar las posibles colisiones de los elementos del juego, y determinar el final de la partida (fin del bucle principal), si es el caso.

- **Renderizado gráfico**. Consiste en representar en pantalla (en la consola) el estado actual del juego. Para ello, serán útiles las instrucciones `Console.Clear()` que limpia la pantalla y `Console.SetCursorPosition(i,j)` que sitúa el cursor en la posición `(i,j)`, de modo que el texto que se escriba a continuación aparecerá a partir de esa posición.

  Para cambiar el color del texto puede utilizarse la instrucción `Console.ForegroundColor = ConsoleColor.Red` (cambia el color a rojo, en este caso). Para restaurar el color se utiliza `Console.ResetColor()`.

- **Tiempo de retardo**. Para que el usuario tenga tiempo de reacción y el juego sea *jugable*, en cada vuelta del bucle debemos introducir el tiempo de retardo `DELTA` con la instrucción `System.Threading.Thread.Sleep(DELTA)`.

A continuación mostramos la estructura general que debe tener el código:

```
using System;
namespace combate {
  class MainClass {
    // declaración de constantes DELTA, ANCHO, ALTO
    ...

    public static void Main(string[] args) {
      Random rnd = new Random(); // generador de aleatorios

      // declaración de constantes y variables para el estado del juego

      // bucle ppal del juego
      while (...) {
        // Input de usuario
        if (Console.KeyAvailable) {
          string dir = (Console.ReadKey(true)).KeyChar.ToString();
          // procesamiento del input de usuario
          ...
        }

        // Lógica del juego: movimiento del enemigo, bomba y bala

        // Control de colisiones

        // Renderizado gráfico
        Console.Clear ();
        // dibujo del borde, enemigo, bomba, jugador, bala

        // retardo
        System.Threading.Thread.Sleep(DELTA);
      } // fin del bucle

    // Información del ganador, despedida...
    }
  }
}
```

Sugerencia: se puede hacer el programa escalonadamente, probando el funcionamiento de cada una de las partes:

- Comenzar declarando las variables y constantes necesarias.
- Hacer el renderizado gráfico para el borde y el jugador, y probarlo (nos permitirá ir depurando posibles errores).
- Recoger el input de usuario y utilizar el renderizado para probar su funcionamiento.
- Añadir la lógica del disparo de balas y su renderizado.
- Añadir la lógica del enemigo y renderizarlo.
- Añadir la lógica de las bombas y renderizarlo.
- Añadir el control de colisiones.
- Probar el juego repetidas veces, variar los valores de `ALTO` y `ANCHO`,...

Fase de pruebas y mejoras

- El programa, tal como está planteado no renderiza el estado inicial del juego (el primer renderizado se hace al final de la primera vuelta del bucle). Sin embargo, la estructura del bucle principal está bien diseñada. ¿Cómo forzamos el renderizado del estado inicial sin tocar la estructura del bucle?
- En el planteamiento propuesto hay fallos en el control de colisiones. ¿Qué fallos son? ¿Cómo pueden corregirse?
- El juego puede hacerse más interesante de varios modos:
  - Puede acelerarse progresivamente la velocidad a medida que avanza el juego.
  - Puede ampliarse el área de movimiento del enemigo de modo que pueda descender más en el área de juego.