Fundamentos de la programación

Práctica 1. Combate aéreo

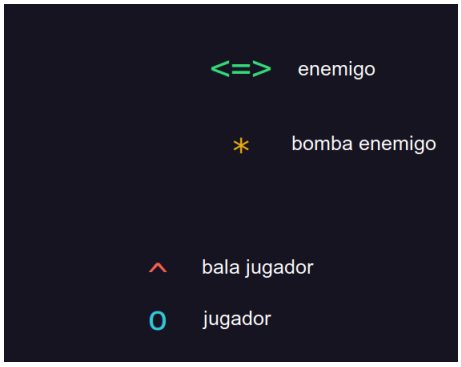
Indicaciones generales:

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

  `// Nombre Apellido1 Apellido2`

- **Lee atentamente** el enunciado y desarrolla el programa tal como se pide, con la representación y esquema propuestos.
- 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 4 de noviembre.

Vamos a implementar un juego marcianitos en consola. Se desarrolla en una cuadrícula rectangular y tiene el siguiente aspecto:



Elementos del juego. El área de juego está determinada por las constantes `FILS` y `COLS` (número de filas y columnas). La esquina superior izquierda tiene coordenadas `(0,0)` y la inferior derecha `(FILS-1,COLS-1)`. El jugador `O` puede moverse por todo el área de juego con las teclas habituales `"asdw"`. También puede disparar balas `^` hacia arriba con la tecla `"L"`. El enemigo `<=>` ocupa tres posiciones en pantalla y se mueve aleatoriamente, pero solo en la mitad superior del área de juego. Además disparará bombas `*` que caen hacia abajo.

Movimiento de entidades. La velocidad de refresco (tiempo entre *frames*) viene dada por una constante `DELTA` (tiempo de retardo entre frames, 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 desde la posición justo 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). Si no hay ninguna bomba en caída, generará un nueva (no puede haber dos bombas en juego simultáneamente). Las bombas arrancarán justo desde la posición inferior al centro del enemigo y se desplazarán una posición hacia abajo en cada frame.

Colisiones. Tanto las balas como las bombas continúan su movimiento hasta salir del área de juego o bien colisionar con otra entidad. Si una bala colisiona con el enemigo, el jugador gana la partida, mientras que si el jugador colisiona con una bomba o con el enemigo, el ganador será el enemigo. También es posible la colisión entre bala y bomba, en cuyo caso se destruyen ambas.

1. Representación interna del juego

Además de las constantes `FILS`, `COLS` y `DELTA`, el *estado interno* del juego vendrá determinado por las posiciones de las entidades implicadas (respecto al origen de coordenadas):

- `(jugFil,jugCol)`: coordenadas del jugador.
- `(eneFil,eneFil)`: coordenadas de *la posición central* del enemigo (solo se almacena esta posición en la representación interna, pero ocupa tres posiciones a todos los efectos).
- `(balaFil,balaCol)`: coordenadas de la bala lanzada por el jugador, si hay bala activa en ese momento. Para indicar que no hay bala activa utilizaremos un valor especial fuera del área de juego: `balaFil = -1`.
- `(bombaFil,bombaCol)`: coordenadas de la bomba lanzada por el enemigo. Como antes, cuando no hay bomba activa tendremos un valor especial `bombaFil = -1`.
- `finPartida`: puede tomar los valores 0 (partida en curso), 1 (partida terminada, gana el jugador), 2 (partida terminada, gana el enemigo).

Todas estas constantes y variables están ya definidas en la plantilla proporcionada en `Program.cs`.

2. Bucle de juego

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

- recogida del input del jugador de teclado.
- movimiento (y generación de entidades, si corresponde) **en este orden**: jugador, bala, enemigo, bomba.
- detección de colisiones.
- renderizado en consola.

Desarrollaremos el juego de manera incremental, añadiendo las distintas entidades con su lógica y su renderizado, y por último añadiremos el control de colisiones. La recogida del input de usuario se da ya implementada en la plantilla y se explica en la siguiente sección.

Para el renderizado serán útiles las instrucciones `Console.Clear()`, que limpia la pantalla, y `Console.SetCursorPosition(i,j)`, que sitúa el cursor en la posición `(left,top)`, de modo que el texto que se escriba a continuación aparecerá a partir de esa posición. Para cambiar el color del texto utilizaremos `Console.ForegroundColor = ConsoleColor.Red` (cambia el color a rojo, en este caso). Para restaurar el color se utiliza `Console.ResetColor()`.

Para que el usuario tenga tiempo de reacción y el juego sea *jugable*, al final de cada vuelta del bucle se incluye la intrucción `System.Threading.Thread.Sleep(DELTA)`, que para la ejecución durante `DELTA` milisegundos.

3. Lectura no bloqueante de teclado (ya implmentada)

Para leer el input de usuario, si hiciésemos la lectura con el habitual `Console.ReadLine()`, la entrada sería *bloqueante*: el programa queda a la espera del input de usuario y la ejecución se bloquea hasta que se pulse *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 (sin esperar intro); si no, continúa la ejecución. Esto puede hacerse del siguiente modo (código incluido en la plantilla):

```
// recogida no bloqueante de INPUT DE USUARIO
string dir="";
if (Console.KeyAvailable) { // si se detecta pulsación de tecla
  // leemos input y transformamos a string
  dir = (Console.ReadKey(true)).KeyChar.ToString();
  // limpiamos buffer para no acumular pulsaciones entre frames
  while (Console.KeyAvailable) Console.ReadKey(true);
}
```

Tras ejecutar este código en la variable `dir` tendremos el input del usuario: si no hay pulsación será `dir=""`; en otro caso, `dir` contendrá la tecla pulsada.

4. Lógica del jugador

Tras leer el input, si `dir` corresponde a alguna de las teclas de movimiento (`"asdw"`), se hará el movimiento correspondiente del jugador, siempre que no se salga del área de juego; en otro caso, no hace nada

En la sección de renderizado, dibujaremos el jugador en pantalla. Antes de continuar comprobaremos que el jugador se renderiza bien y puede moverse libremente sin salirse del área de juego por ninguno de los bordes.

5. Lógica de la bala

Si hay bala activa, esta se desplazará una unidad hacia arriba. Si se sale del área de juego se elimina (haciendo `balaX = -1`). Si no hay bala y el jugador ha pulsado la tecla de lanzamiento, se generará una nueva bala justo encima de la posición del jugador.

En la sección de renderizado, dibujaremos la bala. Comprobaremos que funciona bien el lanzamiento de bala, el movimiento y la salida el área de juego.

6. Lógica del enemigo

El enemigo podrá desplazarse aleatoriamente una posición en cualquier dirección. Para generar números aleatorios se ha inicializado un generador en la plantilla (**el generador solo se inicializará una vez en todo el programa**):

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

Después, cada vez que necesitemos un número aleatorio haremos:

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

Esta instrucción devuelve un entero aleatorio del intervalo *[i,j-1]*, es decir, el valor *j* queda excluido.

Para mover el enemigo, debemos tener en cuenta que solo hemos almacenado su posición central y tendremos que comprobar ninguna de las 3 posiciones que representa se sale del área de juego. En otro caso, 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.

Igual que antes, en la sección de renderizado, dibujaremos el enemigo y comprobaremos su buen funcionamiento.

7. Bombas

Si hay bomba en el área de juego, se desplazará una posición hacia abajo. Si se sale del área de juego se eliminará. Si no hay bomba se generará una nueva justo debajo de la posición central del enemigo.

En la sección de renderizado, dibujaremos la bomba.

8. Colisiones

Una vez aplicada la lógica a cada una de las entidades, comprobaremos las posibles colisiones, asignando a `finPartida` el valor adecuado. Las posibles colisiones son:

- bala-enemigo: termina el juego con victoria del jugador.
- bomba-jugador: termina el juego con victoria del enemigo
- jugador-enemigo: termina el juego con victoria del enemigo

Hay que tener en cuenta el cruce de entidades para no pasar por alto colisiones. Por ejemplo, observemos los dos frames siguientes (consecutivos):

En el primero la bala está justo debajo de la bomba. En el siguiente frame, cada entidad ha avanzado una unidad y se han cruzado, sin detectar la colisión.

Este es un problema clásico en videojuegos y hay distintas aproximaciones para solucionarlo. Reflexiona sobre ello e implementa razonadamente alguna solución limpia y bien estructurada.