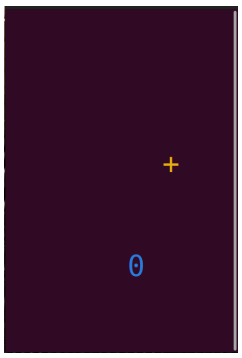
Fundamentos de la programación I

Práctica 1. Juego de la abeja

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 30 de octubre.

Vamos a implementar un sencillo juego en consola en el que tenemos que huir de un abeja que nos persigue. El juego tendrá el siguiente aspecto:



Intervienen dos entidades, el jugador `O` y la abeja `+`. El jugador se controla con las teclas habituales `asdw` y la abeja se mueve de manera autónoma, como veremos. El programa tendrá el aspecto:

```
1  namespace Main {
2    class MainClass {
3      static Random rnd = new Random(); // generador de aleatorios (para mover abeja)
4      const int ANCHO = 12, ALTO = 9;   // dimensiones del área de juego
5
6      public static void Main (){
7        Console.SetWindowSize(ANCHO,ALTO); // tamaño de la consola
8        int jugF, jugC,         // posición del jugador
9            abejaF, abejaC,     // posición de la abeja
10           delta = 300;        // retardo entre frames (ms)
11       bool colision = false; // colisión entre abeja y jugador
```

El **estado del juego** queda determinado por:

- Las constantes `ANCHO` y `ALTO`, que determinan el área de juego.
- `jugF, jugC`: posición del jugador (fila y columna).
- `abejaF, abejaC`: posición de la abeja
- `delta`: tiempo de retardo entre frames (en milisegundos)
- `colision`: determina si la abeja ha alcanzado al jugador.

Las posiciones del jugador y la abeja son relativas a la **esquina superior izquierda del área de juego** $(0, 0)$. Se inicializarán de modo que estén suficientemente alejados (por ejemplo uno en el tercio superior izquierdo y otro en el inferior derecho).

1. Versión preliminar del juego

Después de las declaraciones anteriores en el programa tendremos el bucle principal con el siguiente esquema, que habrá que completar **sin alterar el orden de las acciones indicadas**:

```
1  while (...){  // mientras no termine el juego
2    // recogida de de input
3    string s="";
4    while (Console.KeyAvailable) s = (Console.ReadKey(true)).KeyChar.ToString();
5
6    // movimiento del jugador (en función del input)
7    ...
8
9    // movimiento aleatorio de la abeja
10   ...
11
12   // detección de colisión
13   ...
14
15   // renderizado de las entidades en consola
16   ...
17
18   // retardo entre frames
19   System.Threading.Thread.Sleep(delta);
20 }
```

En cada vuelta del bucle se hacen las siguientes acciones: lectura del input, movimiento del jugador, movimiento de la abeja, detección de colisión, renderizado y retardo (ya implementado).

La primera acción es la recogida del input que se da ya implementada en las líneas 3 y 4. Estas instrucciones hacen *lectura no bloqueante* de teclado: si se pulsa una tecla, se lee y se guarda su valor en `s`; si no hay pulsación la ejecución continúa con (`s=""`). En cualquier caso la ejecución no se para como ocurriría con la lectura (bloqueante) habitual `Console.ReadLine()`. El `while` de línea 4 funciona como sigue: si se pulsa más de una tecla en el mismo frame se *consumen todas las pulsaciones*, pero se guarda como input solo la última de ellas... ¿por qué es conveniente este comportamiento?

El programa debe desarrollarse de manera incremental en el siguiente orden:

- Movimiento del jugador: se procesa el input leído en `s` moviendo el jugador en función de la tecla pulsada (`asdw`). Si se encuentra en un borde del área de juego y se intenta un movimiento en la dirección de ese borde no se mueve en ese frame.
- Renderizado del jugador: se escribe en pantalla el jugador en su posición en azul. Serán útiles los métodos:
  - `Console.Clear()`: limpia la pantalla (antes de empezar un nuevo renderizado).
  - `Console.SetCursorPosition(left,top)`: sitúa el cursor en la columna `left` y fila `top`, sabiendo que `(0,0)` es la esquina superior izquierda.
  - `Console.ForegroundColor = ConsoleColor.Blue`: cambia a azul el color del texto (análogo con otros colores `Yellow`, `Red`, etc).

  En este punto debe comprobarse que el jugador se renderiza bien, responde bien a las teclas y respeta los límites del área de juego.
- Movimiento de la abeja: se mueve aleatoriamente en alguna de las 4 direcciones básicas. Igual que en el jugador, el movimiento no se realiza si se intenta traspasar alguno de los bordes.

Para obtener una dirección aleatoria utilizaremos el generador `rnd` declarado al principio. La expresión `rnd.Next(a,b)` genera un entero aleatorio del intervalo `[a,b)` (cerrado-abierto!).

- Renderizado de la abeja: se extenderá el renderizado previo para mostrar la abeja en su posición, en amarillo.
- Detección de colisiones: comprueba si la abeja y el jugador están en la misma posición, actualizando la variable `colision`.
- Renderizado de colisiones: se modifica/extiende el renderizado anterior de modo que si hay colisión se escribirá el carácter `*` (en rojo) en la posición de las entidades (cuando hay colisión están en la misma posición).

El bucle debe terminar cuando se detecte colisión entre las entidades. Al final de cada renderizado, el cursor queda situado en pantalla a continuación del último carácter que se haya escrito, lo que puede resultar poco estético. Puede ocultarse el cursor poniendo la instrucción `Console.CursorVisible = false;` en el método `Main`, al principio.

2. Mejoras del juego

Una primera mejora sencilla es permitir al jugador abortar el juego con la tecla `"q"`. Esto es fácil de implementar extendiendo el procesamiento del input y utilizando otro booleano para controlar la terminación del bucle. Vamos a ver algunos refinamientos más elaborados.

2.1. Renderizado

Tal como está planteado el esquema de juego el estado inicial del juego no se renderiza: el primer renderizado se hace al final del bucle, cuando ya se ha hecho una transición de estado. Para mostrar ese estado hay que hacer un renderizado justo antes de entrar en el bucle principal[1].

Para ello basta con copiar el código de renderizado y pegarlo justo antes del bucle principal. ¿Podríamos evitar esta duplicidad de código de alguna manera sencilla y estructurada, renderizando todos los estados del juego?

2.2. Afinando el control de colisiones

La implementación anterior presenta un problema con las colisiones, que puede ilustrarse con la siguiente secuencia de frames:

En el frame de la izquierda se tiene al jugador una posición a la derecha de la abeja. Supongamos que el jugador decide moverse a la izquierda, mientras que la abeja se mueve a la derecha. Tal como está implementado el juego ambos movimientos se hacen de manera simultánea (en una misma iteración del bucle principal) y el efecto es que las entidades se cruzan sin detectar la colisión, tal como se muestra a la derecha.

Este es un problema clásico en videojuegos y puede abordarse de distintos modos. En nuestro caso, modificaremos el código del bucle principal para *comprobar las colisiones cada vez que se mueve una entidad*: se mueve el jugador y se comprueban colisiones; si no hay colisión se mueve la abeja y vuelven a comprobarse las colisiones. De este modo, en la situación de arriba, cuando se mueve el jugador a la izquierda se detecta la colisión y se para el juego. ¿Esta solución es generalizable a cualquier

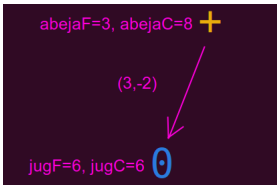
[1] Podría hacerse un solo renderizado justo al principio del bucle principal, pero entonces no mostraríamos el estado final del juego. Por metodología, haremos siempre renderizado antes de entrar en el bucle y al final del mismo.

situación de nuestro juego? ¿Es generalizable a cualquier videojuego en el que hay movimiento de entidades y colisiones?

2.3. Una IA sencilla para la abeja

El movimiento aleatorio de la abeja plantea poca dificultad para el jugador y el juego resulta demasiado sencillo. Para hacerlo más interesante podemos hacer que la abeja *persiga* al jugador, i.e., que se mueva en dirección al jugador.

Para ello, en primer lugar calcularemos el vector de dirección entre la abeja y el jugador. Luego escogeremos la dirección básica (de las 4 posibles) que mejor aproxime a dicho vector y moveremos la abeja en esa dirección teniendo en cuenta los bordes. En el estado del juego mostrado al principio tendremos:



Para el vector calculado *(3,-2)*, la dirección básica que mejor aproxima es *(1,0)*, es decir, la abeja se moverá hacia abajo. Este cálculo puede realizarse de manera sencilla.

Una vez implementada esta IA veremos que la abeja se vuelve implacable y acorrala al jugador con mucha facilidad: el juego es prácticamente *injugable*. Una forma sencilla de relajar la situación y *calibrar* el juego es hacer que la abeja sea más lenta, haciendo que se mueva solo cada 2 frames. ¿Queda así la dificultad del juego bien equilibrada? ¿Qué otras alternativas pueden funcionar?