Fundamentos de la programación 2

Práctica 1. Naves

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 los métodos, parámetros y requisitos que se especifican. No puede modificarse la representación propuesta, ni alterar los parámetros de los métodos especificados, a excepción de la forma de paso (`out`, `ref`, ...), que debe determinar el alumno.

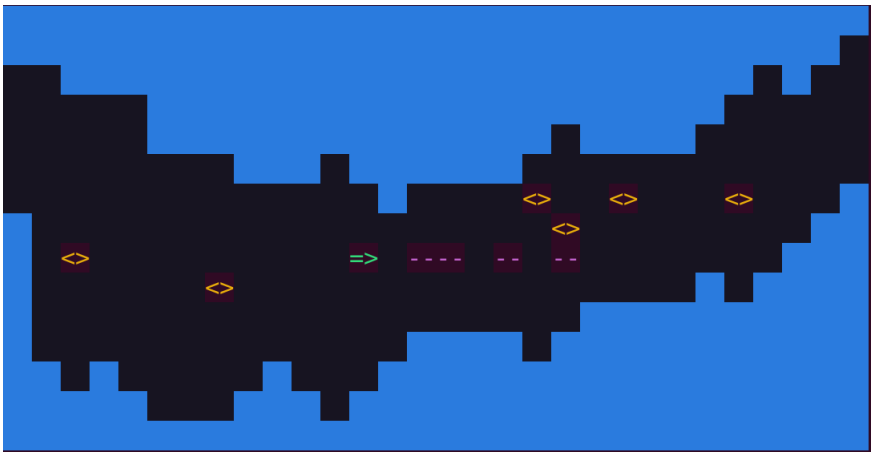
  Pueden implementarse todos los métodos adicionales que se consideren oportunos, especificando claramente su cometido, parámetros, etc.

- 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 límite para la entrega** es el 8 de marzo.

En esta práctica vamos a implementar un juego para pilotar una nave a través de un túnel, disparando para destruir los enemigos que aparecen. El aspecto en consola es este:



La flecha `=>` representa la nave que pilotamos, la zona negra es el túnel por donde avanza, la azul es muro sólido con el que no debemos colisionar, los ángulos `<>` representan enemigos y los guiones `--` son las balas que disparamos. En cada frame la nave avanza una posición en el túnel: desaparece la columna de la izquierda y aparece una nueva a la derecha como continuación del túnel. Si no pulsamos ningún cursor, la nave se queda en la misma posición del área de juego (se percibe como que avanza una posición en el túnel). Si se pulsa un cursor la nave se mueve una posición en la dirección correspondiente. El movimiento está limitado por los extremos laterales visibles del túnel y por los límites verticales del túnel (con los que puede colisionar). Además en cada frame la nave podrá lanzar una bala (con `'x'` o espacio) que avanzará en línea recta hacia adelante hasta colisionar con un enemigo (lo destruye), o el muro (destruye esa parte del túnel), o salir del área visible por la derecha. Los enemigos permanecerán estáticos respecto al túnel hasta desaparecer por la izquierda o ser destruidos por una bala.

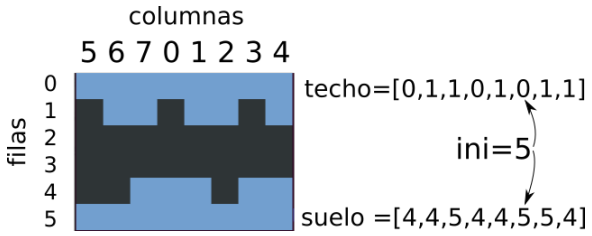
Se proporciona una plantilla en `Program.cs` con algunos métodos ya implementados y procederemos paso a paso, añadiendo elementos de manera incremental. **Es muy importante comprobar el buen funcionamiento en cada etapa antes de avanzar a la siguiente**.

1. Túnel: renderizado, avance e inicialización

El área de juego está determinada por las constantes `ANCHO` (número de columnas) y `ALTO` (número de filas), declaradas al principio (fuera de los métodos). El túnel está definido del siguiente modo:

```
struct Tunel {
    public int [] suelo, techo;
    public int ini;
}
```

Los arrays de enteros `suelo` y `techo` (de tamaño `ANCHO`) determinan respectivamente las posiciones del suelo y el techo del túnel (siempre relativas a la fila superior, fila 0). Por eficiencia, los manejamos como **arrays circulares**: el campo `ini` representa la primera columna del tunel, `ini+1` la siguiente y así sucesivamente, hasta llegar a `ANCHO-1`; después se continúa con las posiciones 0,1,2... hasta `ini-1`, que será la última columna (salvo que `ini=0`, en cuyo caso la última columna estaría en 0). Por ejemplo, para `ANCHO=8` y `ALTO=6`, podríamos tener `ini=5`, `techo = [0,1,1,0,1,0,1,1,]` y `suelo = [4,4,5,4,4,5,5,4]`. Gráficamente:



Con esta representación circular, dado un índice `i`:

- el **siguiente** elemento está en `(i+1) % ANCHO` y el **anterior** en `(i+ANCHO-1) % ANCHO`

Para la gestión del túnel se proporcionan ya implementados los siguientes métodos en la plantilla:

- `void AvanzaTunel(ref Tunel tunel)`: consiste en desplazar `ini` una unidad a la derecha cíclicamente de modo que (virtualmente) el túnel avanza hacia la izquierda. La posición `ini` pasa a representar la última columna, para la que generamos nuevos valores para el suelo y el techo. El algoritmo implementado garantiza escalones suaves en el túnel. Este método es más eficiente implementado con arrays circulares que con arrays normales, ¿por qué?

- `void IniciaTunel(out Tunel tunel)`: crea los arrays `suelo` y al `techo`, inicializa la primera columna y después invoca `ANCHO-1` veces al método `AvanzaTunel` para generar el resto.

¿Por qué `tunel` pasa como `ref` en `AvanzaTunel` y como `out` en `IniciaTunel`?

A continuación implementaremos el renderizado del túnel:

- `void RenderTunel(Tunel tunel)`: dibuja en pantalla el túnel. Para ello recorremos las columnas empezando en `tunel.ini` dibujando el suelo y el techo de cada una. El avance en las columnas se calcula con la fórmula para el **siguiente** mencionada más arriba.

  Utilizaremos el método `Console.SetCursorPosition(left,top)` para situar el cursor en la posición adecuada de la pantalla. Para proporcionar el aspecto gráfico, cada cuadrado del túnel ocupará dos caracteres en pantalla (al igual que el resto de entidades).

  En general, un elemento en (i, j) en la representación interna, en pantalla ocupará las dos posiciones $(2{*}i, j)$ y $(2{*}i{+}1, j)$. **Esto solo afecta al renderizado; en el resto del programa las entidades están determinados por una posición simple `(i,j)`, sin duplicidad.**

Depuración: será útil declarar una constante booleana `DEBUG`. En el renderizado, si `DEBUG=true` escribiremos en pantalla por debajo del área de juego el valor de `ini`, así como los arrays `suelo` y `techo` para verificar que todo funciona correctamente. Cuando el programa esté bien depurado, haciendo `DEBUG=false` se puede omitir toda esa información. Asegurarse de que el renderizado del túnel funciona correctamente llamando a `RenderTunel` desde `Main` con el ejemplo de arriba y algunos otros introducidos manualmente con valores pequeños de `ANCHO`.

Ya podemos implementar una primera versión del método `Main`: inicializar y renderizar el túnel; implementar el bucle principal: avanzar y renderizar el túnel en cada iteración.

2. Entidades del juego: generación y eliminación

Para representar las entidades del juego (nave, enemigos, balas) utilizaremos el siguiente tipo:

```
struct Entidad {
  public int fil,col;
}
```

donde `fil` y `col` representan la posición de la entidad en cuestión. Para los grupos de entidades (balas, enemigos y colisiones) utilizaremos el tipo:

```
struct GrEntidades {
    public Entidad [] ent;
    public int num;
}
```

que consta de un array de entidades `ent` y un entero `num` que determina el número de entidades que contiene dicho array. El número máximo de enemigos y de balas viene determinado por las constantes `MAX_BALAS` y `MAX_ENEMIGOS`. El array de entidades `ent` no tiene ningún orden en particular y los elementos ocuparán las posiciones iniciales `[0..num-1]`. De este modo, el valor `gr.num` coincide con la primera posición libre en el array.

Para añadir y eliminar entidades implementamos los métodos:

- `void AñadeEntidad(Entidad ent, GrEntidades gr)`: añade la entidad `ent` a la estructura `GrEntidades`, i.e., copia la entidad `ent` en la última posición de `gr.ent` e incrementa `gr.num`.
- `void EliminaEntidad(int i, GrEntidades gr)`: elimina la entidad `i`-ésima de `gr`. Una forma eficaz de hacerlo es copiar la última entidad del array en la posición `i` y decrementar `gr.num`.

3. Nave: inicialización, avance y renderizado

La nave será una variable `nave` de tipo `Entidad`. Inicialmente se posiciona en la columna central (`ANCHO/2`), centrada respecto al suelo y al techo en dicha columna. Para eliminar la nave (cuando incorporemos colisiones) haremos `nave.fil=-1`. Para el movimiento de la nave implementamos:

- `void AvanzaNave(char ch, Entidad nave)`: la posición de `nave` cambia en función de `ch`:
  - `'l'/'r'`: se decrementa/incrementa la columna, siempre que no se salga del área de juego (en otro caso no se modifica).
  - `'u'/'d'`: se decrementa/incrementa la fila.
  - en otro caso, no se hace nada.

Para el visualizar el tunel y la nave en pantalla (y más adelante el resto de entidades) implementaremos un método `Render`:

- `void Render(Tunel tunel, Entidad nave)`: invoca al método anterior `RenderTunel` para dibujar el túnel y después dibuja la nave (si hay nave) con la flecha correspondiente. En modo `DEBUG` se escribirá también la posición de la nave fuera del área de juego.

Para poder mover la nave desde el juego, necesitaremos el método `LeeInput` (ya implementado en la plantilla), que gestiona la lectura de teclado: devuelve `'l','r','u','d'` para las direcciones[1].

Ahora extenderemos el método `Main` para inicializar la posición de la nave y después en el bucle principal leer el input de usuario y mover la nave de acuerdo al mismo.

Importante:

- Toda la lectura de teclado se hará desde `LeeInput`. Ningún otro método lee nada de teclado.
- Todo el renderizado (toda la escritura en pantalla) se hará exclusivamente a través de los métodos `Render/RenderTunel`. Ningún otro método escribirá nada en pantalla.

4. Enemigos: generación, avance y renderizado

Los enemigos se guardarán en una variable `enemigos` de tipo `GrEntidades`. Inicialmente no hay enemigos (`enemigos.num=0`) y para gestionarlos tendremos:

- `void GeneraEnemigo(GrEntidades enemigos, Tunel tunel)`: si no se ha alcanzado el máximo número de enemigos en juego, se genera uno nuevo con una probabilidad del 25 %, situado a la derecha del área de juego, en una fila aleatoria entre el suelo y el techo de esa columna.
- `void AvanzaEnemigos(GrEntidades enemigos)`: hace avanzar a cada uno de los enemigos una posición a la izquierda, eliminando aquellos que se salen del área de juego (por la izquierda).

  Pista: aquí puede producirse un error típico de programación. Al recorrer una estructura eliminando elementos, el tamaño la misma puede disminuir en cada iteración alterando el propio recorrido.

Para el visualizar los enemigos en pantalla extenderemos el método `Render` de la sección **??**, que ahora recibirá la variable `enemigos` y los dibujará en pantalla. Extenderemos también el bucle principal que llamará a (por este orden): `LeeInput`, `AvanzaTunel`, `GeneraEnemigo`, `AvanzaEnemigos`, `AvanzaNave`, `Render`, `Thread.Sleep(100)`. Al ejecutar deberemos ver el avance del túnel y los enemigos, que deben generarse y desaparecer según lo explicado; además podremos mover la nave.

5. Balas: generación, avance y renderizado

Las balas se definen con una variable `balas` de tipo `GrEntidades`, con `balas.num=0` incialmente. Implementaremos los métodos:

- `void GeneraBala(GrEntidades balas, Entidad nave)`: si no se ha alcanzado el máximo número de balas en juego y la nave no está pegada al extremo derecho del área de juego, se genera una nueva bala **justo en la posición de la nave**; en otro caso no hace nada.

[1] Además devuelve `'x'` para el lanzamiento de balas, `'p'` para pausar y `'q'` para abortar juego, que se usarán más adelante.

- `void AvanzaBalas(GrEntidades balas)`: para cada una de las `balas` avanza su posición una unidad a la derecha; si se sale del área de juego, la elimina de la estructura (tener en cuenta la misma pista que en `AvanzaEnemigos`).

Para el renderizado, extendereremos el método `Render` con el parámetro `balas` y las dibujaremos en pantalla guiones morados. En el bucle principal, después de `AvanzaNave` incluiremos una llamada a `GeneraBala` si el input leído es `'x'`; y después llamaremos a `AvanzaBalas`, justo antes del renderizado.

6. Colisiones

Declararemos la variable `colisiones` de tipo `GrEntidades` para almacenar las colisiones. La idea es que cuando se produce una colisión entre dos entidades, dichas entidades se eliminan de sus correspondientes estructuras y se añade una nueva colisión en la estrucutura `colisiones`. Las colisiones se mostrarán en pantalla durante un frame y después se eliminarán.

Implementaremos los siguientes métodos:

- `ColNaveTunel(Tunel tunel, Entidad nave, GrEntidades colisiones)`: detecta y gestiona una posible colisión entre la nave y el túnel. Como el túnel utiliza arrays circulares, en primer lugar calcularemos el índice del túnel correspondiente a la columna de la nave. Si hay colisión con el suelo o el techo, se elimina la nave y se añade una colisión a `colisiones`.
- `void ColBalasTunel(Tunel tunel, GrEntidades balas, GrEntidades colisiones)`: para cada una de las balas en juego comprueba si hay colisión con el suelo o el techo. En ese caso elimina la nave y los bloques correspondientes en el túnel, y añade una colisión en la posición de la nave.
- `void ColNaveEnemigos(Entidad nave,GrEntidades enemigos,GrEntidades colisiones)`: para cada uno de los enemigos en juego comprueba si hay colisión con la nave, en cuyo caso elimina la nave y el enemigo correspondiente, y genera una nueva colisión en esa posición.
- `void ColBalasEnemigos(GrEntidades balas, GrEntidades enemigos, GrEntidades colisiones)`: para cada uno de los enemigos comprueba si hay colisión con cada una de las balas; en ese caso elimina el enemigo y la bala correspondientes, y añade una nueva colisión.
- `Colisiones(Tunel tunel, Entidad nave, GrEntidades balas, GrEntidades enemigos, GrEntidades colisiones)`: invoca a cada uno de los 4 métodos anteriores.

Ahora el método de renderizado quedará como `void Render(Tunel tunel, Entidad nave,GrEntidades balas, GrEntidades enemigos, GrEntidades colisiones)` y mostrará en pantalla el túnel y todas las entidades de juego.

En el método `Main`, para incorporar la gestión de colisiones tenemos un problema que resolver (clásico en videojuegos) que se puede ilustrar como sigue:

Cuando una bala está a punto de colisionar con el túnel (primera imagen), si en un mismo frame la bala avanza a la derecha y el tunel a la izquierda se cruzarían (segunda imagen) sin detectar la colisión. Esto mismo puede ocurrir con la nave y el túnel, o con una bala y un enemigo.

En nuestra implementación, este problema puede solucionarse intercalando dos llamadas a `Colisiones` en cada frame. Haremos:

- Llamar a `AvanzaTunel, GeneraEnemigo, AvanzaEnemigos`

- Comprobar `Colisiones`:
  - Si la nave no ha colisionado llamar a `AvanzaNave`, `GeneraBala` (si corresponde), `AvanzaBalas` y volver a comprobar `Colisiones`
  - En otro, caso no hacer nada.
- Llamar a `Render`, `Sleep` y eliminar todas las colisiones.

El bucle principal terminará cuando la nave colisione o se aborte la partida.

Por último, incorporar las **opciones de salvar y restaurar partida** utilizando un archivo de texto.

7. Extensiones opcionales

- Incorporar música y sonidos para el disparo de balas y las colisiones.
- Implementar animaciones para las colisiones de modo que cuando una bala destruya más de un bloque se muestren todos los bloques destruidos en rojo durante algunos frames.