Fundamentos de la programación I

Práctica 2. Naves

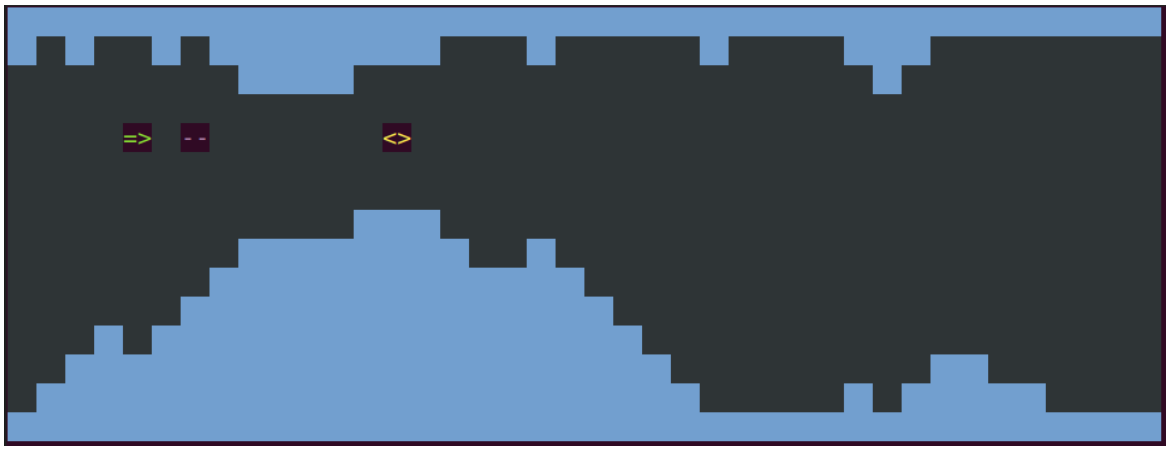
Indicaciones generales:

- 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 la representación y esquema propuestos, implementando los métodos que se especifican, **respetando los parámetros y tipos de los mismos**. Pueden implementarse los métodos auxiliares que se consideren oportunos comentando 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 de entrega** finaliza el lunes 9 de diciembre. **Se anima a presentar la semana anterior para poder corregir y mejorar el programa si fuera necesario.**

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

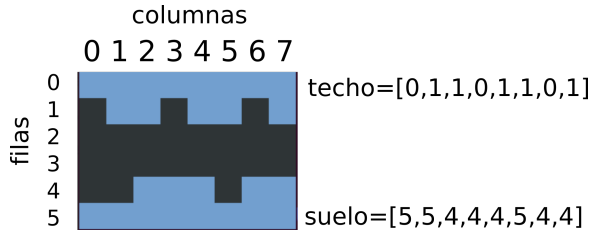


La flecha => representa la nave que pilotamos, la zona gris es el túnel por donde avanza la nave, la azul es muro sólido, los ángulos <> representan el enemigo y los guiones -- la bala lanzada hacia adelante por nuestra nave. En cada frame avanzamos una posición en el túnel: desaparece la columna de 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 pulsamos el cursor izquierdo/derecho retrocede/avanza una posición. El movimiento de la nave está limitado por los extremos laterales visibles del túnel y por los límites verticales del túnel (con el que puede colisionar). Si pulsamos arriba/abajo la nave sube/baja una posición. Además en cada frame la nave podrá lanzar una bala que avanzará en línea recta hacia adelante hasta colisionar con el enemigo (lo destruye) o el muro (destruye el cuadrado con el que colisiona). A lo sumo puede haber un enemigo en pantalla, que permanecerá estático respecto al túnel hasta desaparecer por la izquierda o ser destruido por una bala. Y solo puede haber una bala, que desaparece al colisionar o perderse por la derecha del área visible del túnel.

Para desarrollar la práctica se proporciona una plantilla de partida en `Program.cs` con algunos métodos ya implementados y procederemos paso a paso, añadiendo elementos de manera incremental. **Es importante comprobar el buen funcionamiento en cada etapa antes de avanzar a la siguiente**. En el prototipo de los metodos se especifican los parámetros, pero no la forma de paso (valor, ”out”, ”ref”).

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 por dos arrays de enteros: `suelo` y `techo`, ambos de tamaño `ANCHO` y cuyos valores determinan respectivamente las posiciones del suelo y el techo del túnel (siempre relativas a la fila superior, fila 0). Por ejemplo, para `ANCHO=8` y `ALTO=6`, podríamos tener `suelo = [5,5,4,4,4,5,4,4]` y `techo = [0,1,1,0,1,1,0,1]`. Gráficamente:



Para la gestión del túnel implementaremos los siguientes métodos:

- `void RenderTunel(int [] suelo, int [] techo)`: dibuja en pantalla el túnel: se recorren las columnas dibujando, el suelo y el techo de cada una de ellas. Utilizaremos el método `Console.SetCursorPosition(left,top)` para situar el cursor en la posición adecuada. Para escribir cuadrados azules se cambia el color de fondo con `Console.BackgroundColor = ConsoleColor.Blue` y luego se escriben blancos `" "`.

  Importante: para proporcionar el aspecto gráfico, cada cuadrado del túnel ocupará dos caracteres en pantalla (igual que la nave, el enemigo y la bala). **Esto solo afecta al renderizado; en el resto del programa todos los elementos (cuadrados del techo y suelo, nave, enemigo, bala) están determinados por una posición simple `(i,j)`, sin duplicidad.**

  Probar el renderizado y asegurarse de que funciona correctamente llamando a `RenderTunel` desde `Main` con el ejemplo de arriba y otros introducidos manualmente.

- `void AvanzaTunel(int [] suelo, int [] techo)`. Este método se da **ya implementado** en la plantilla. Consiste en desplazar todos los elementos del suelo y el techo una posición a la izquierda (desaparece la primera posición) y generar un nuevo valor para la última posición de ambos arrays, que aleatoriamente incremente o decremente el valor previo.

- `void IniciaTunel(int [] suelo,int [] techo)`: para generar el túnel de partida se inicializa la última columna con `techo[ANCHO-1]=0; suelo[ANCHO-1]=ALTO-1;` y después se invoca `ANCHO-1` veces al método `AvanzaTunel`.

Depuración: es útil declarar una constante booleana `DEBUG`. En el renderizado, si `DEBUG=true` escribiremos en pantalla por debajo del área de juego el contenido de `suelo` y `techo` para verificar que todo funciona correctamente. Cuando el programa esté bien depurado haciendo `DEBUG=false` se omite toda esa información.

2. Método `Main` y bucle principal de juego: primera versión

Ya podemos implementar una versión inicial de método `Main`. En primer lugar se inicializa el túnel y se renderiza. A continuación vendrá el bucle principal. En cada iteración, por ahora lo único que hace es avanzar el túnel, renderizarlo y poner un retardo de tiempo (120ms es adecuado). En pantalla debe verse el avance del túnel.

A medida que vayamos introduciendo elementos en el juego iremos ampliando el renderizado y el método `Main`.

3. Enemigo: generación, avance y renderizado

El enemigo viene dado por sus coordenadas `enemigoC, enemigoF` (columna y fila), siendo `enemigoC=-1` cuando no hay enemigo en juego (en ese caso da igual el valor de `enemigoF`). Inicialmente no hay enemigo. Para gestionarlo implementamos el método:

- `void GeneraAvanzaEnemigo(int enemigoC, int enemigoF, int[] suelo, int[] techo)`: si no hay enemigo en juego lo genera con una probabilidad de 1/4, situado en el extremo derecho del área de juego y en una fila aleatoria dentro del túnel (entre el suelo y el techo de esa columna). Si ya hay enemigo lo mueve una posición a la izquierda (se percibe estático respecto al túnel).

Para el renderizado del estado del juego implementaremos un método `Render`. Por ahora solo renderizará el túnel y el enemigo, y posteriormente lo extenderemos con el resto de entidades:

- `Render(int [] suelo, int [] techo, int enemigoC, int enemigoF)`: invoca al método anterior `RenderTunel` para dibujar el túnel y después, si hay enemigo en juego, lo dibuja. Como ya hemos dicho, para el renderizado hay que tener en cuenta que **cada coordenada interna del juego se corresponde con 2 caracteres en pantalla**. Por ejemplo, si `(enemigoC,enemigoF)=(3,4)`, en pantalla se dibujarán los ángulos <> en sendas posiciones `(6,4)` y `(7,4)`. En general, un elemento en las coordenadas (i, j), en pantalla aparecerá en las posiciones $(2 * i, j)$ y $(2 * i + 1, j)$.

  En modo `DEBUG` se escribirá también la posición del enemigo fuera del área de juego.

Importante: 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.

Ahora extenderemos el bucle principal que llamará a (por este orden): `AvanzaTunel`, `GeneraAvanzaEnemigo`, `Render`, `Sleep`, y al ejecutar deberemos ver el avance del túnel y del enemigo (que debe desaparecer y regenerarse según lo explicado).

4. Nave: inicialización, avance y renderizado

La nave queda definida con sus coordenadas `naveC, naveF` (columna y fila). Se posicionará inicialmente en la columna central (`ANCHO/2`), centrada respecto al suelo y al techo en dicha columna. Por defecto, si no se tocan los cursores, la nave se queda en la misma posición (como el túnel se mueve a la izquierda la nave avanza respecto al túnel). El avance de la nave se implementa con el método:

- `void AvanzaNave(char ch, int naveC, int naveF, int enemigoC, int enemigoF, int [] suelo, int [] techo)`:
  dado el carácter `ch`, la posición del enemigo (`enemigoC, enemigoF`), el suelo y el techo, la posición de la nave (`naveC, naveF`) se cambia como sigue: si la nave está sobre el enemigo o sobre el suelo o el techo no se modifica (esto facilitará después el control de colisiones); en otro caso, dependiendo del valor 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 (más adelante se detectará la colisión con el túnel).
  - en otro caso, no se hace nada.

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] Para el renderizado, añadiremos la posición de la nave al método `Render` de la Sección 3 y la dibujaremos en pantalla con la flecha verde (en modo `DEBUG` mostraremos también sus coordenadas como hicimos con el enemigo).

Importante: toda la lectura de teclado se hará desde el método `LeeInput`. Ningún otro método hace ninguna lectura de teclado.

5. Bala: generación, avance y renderizado

La bala viene dada por sus coordenadas `balaC, balaF` (columna y fila), siendo `balaC=-1` cuando no hay bala en juego. La gestión se hace con el método:

- `void GeneraAvanzaBala(char ch, int balaC, int balaF, int naveC, int naveF, int enemigoC, int enemigoF, int [] suelo, int [] techo)`:

  dados `ch`, la posición de la nave `(naveC,naveF)`, el `suelo` y el `techo` hace lo siguiente:

  - si `ch=='x'` y no hay bala en juego, genera una nueva bala justo por delante de la nave.
  - si hay bala en juego y no está sobre el túnel ni sobre el enemigo, avanza una posición hacia la derecha (si está sobre el túnel o el enemigo no se toca para facilitar después el control de colisiones).

Para el renderizado, añadiremos la posición de la bala al método `Render` de la Sección 3 y la dibujaremos en pantalla con los guiones morados (en modo `DEBUG` escribiremos también sus coordenadas).

6. Control de colisiones para nave y bala

La nave puede colisionar contra el enemigo o contra el túnel. Para gestionarlo, implementamos el método:

- `bool ColisionNave(int naveC, int naveF, int [] suelo, int [] techo, int enemigoC, int enemigoF)`:

  determina si la nave está en colisión contra el suelo, el techo o el enemigo (con la representación propuesta es una comprobación muy simple).

Por su parte, la bala puede salirse del área de juego (por la derecha), colisionar contra el enemigo, el suelo o el techo. Cuando colisiona, debe destruir el elemento alcanzado y devolver la posición de la colisión. El método queda como sigue:

- `void ColisionBala(int balaC, int balaF, int enemigoC, int enemigoF, int [] suelo, int [] techo, int colC, int colF)`:

  dadas las posiciones de la bala, el enemigo, el suelo y el techo:

  - si la bala se sale por la derecha la elimina del juego haciendo `balaC=-1`.
  - si colisiona con el enemigo elimina del juego la bala y el enemigo y devuelve en `(colC,colF)` la posición de colisión.

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

- si colisiona con el suelo o el techo elimina del juego la bala y devuelve en `(colC,colF)` la posición de colisión. Además elimina los bloques del tunel correspondientes: si impacta con un bloque del techo elimina ese y todos los bloques por debajo del mismo; si impacta con un bloque del suelo elimina ese y todos los que quedan por encima. Por ejemplo, en el siguiente caso eliminaría 3 bloques del techo:

- si no hay colisión simplemente devuelve `colC=colF=-1` para indicarlo.

7. Programa principal y renderizado final

El renderizado final tendrá el siguiente aspecto:

```
static void Render(int [] suelo, int [] techo, // tunel
                   int naveC, int naveF, int balaC, int balaF, // nave y bala
                   int enemigoC, int enemigoF, // enemigo
                   bool crashNave,             // colisión nave
                   int colC, int colF){        // colisión bala
    // renderizado de tunel ...
    // renderizado de enemigo ...
    // renderizado de nave con o sin colisión ...
    // renderizado de bala con o sin colision ...
}
```

Y sacará en pantalla todas las entidades, incluidas las colisiones para las que dibujará ** en la posición correspondiente (tanto en la colisión con el túnel como con el enemigo).

Y el programa principal tendrá el siguiente aspecto (sin control de colisiones):

```
const bool DEBUG=true;
const int ANCHO=30, ALTO = 15;
static Random rnd = new Random();

static void Main(){
    int [] suelo = new int[ANCHO], // límites del tunel
           techo = new int[ANCHO];

    int naveC, naveF, balaC, balaF, enemigoC, enemigoF;
    // incialización nave, bala, enemigo, túnel
    ...
    Render(...)

    // bucle ppal
    while (...){
        char ch = LeeInput();
        if (ch=='q') ...      // terminar
        else if (ch=='p') ... // pausa
        else {
            AvanzaTunel(...)
            GeneraAvanzaEnemigo(...)
            AvanzaNave(...)
            GeneraAvanzaBala(...)
            Render(...)
            Thread.Sleep(120);
        } //else
    } // while
} // Main
```

Para hacer una buena gestión de colisiones aplicaremos la técnica explicada en clase: cada vez que se mueve una entidad comprobamos sus posibles colisiones antes de continuar. Si colisiona con otra entidad, esta segunda ya no se mueve.

El juego admite infinidad de extensiones y variantes. Hablar con los profesores antes si se quiere implementar cualquiera de ellas.