Fundamentos de la programación

Grado en Desarrollo de Videojuegos

Examen final. Convocatoria extraordinaria de julio. Curso 20-21

Indicaciones generales:

- Se entregará únicamente el archivo `Program.cs` con el programa pedido.
- Las líneas 1 y 2 serán comentarios de la forma:
  `// Nombre Apellido1 Apellido2`
  `// Laboratorio, puesto`
- **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. No se especifica el modo de paso de los parámetros en los métodos (`out`, `ref`, ...), que debe determinar el alumno.

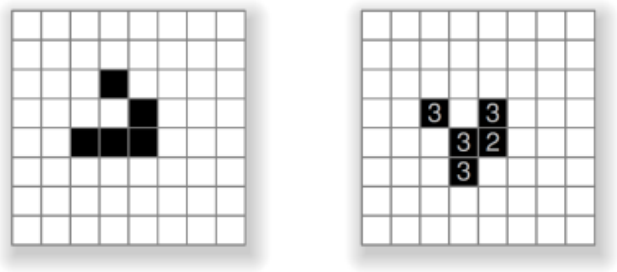
  Pueden implementarse todos los métodos adicionales que se consideren oportunos, especificando claramente su cometido, parámetros, etc.
- **El programa debe compilar y funcionar correctamente, y estar bien estructurado y comentado**. Se valorarán la claridad, la concisión y la eficiencia.
- La **entrega:** se realizará a través del servidor FTP de los laboratorios.

  Importante: comprobar el archivo entregado en el puesto del profesor.

En este ejercicio vamos a implementar el **juego de la vida**, una simulación minimalista de vida artificial desarrollada por John Horton Conway en 1970. Consiste en un entramado de casillas en las que se desarrolla la vida de una colonia de células, gobernada por tres sencillas reglas que determinan la siguiente generación:

- **Regla de Supervivencia**: cada célula que tenga **2 ó 3** vecinos sobrevive y pasa a la generación siguiente.
- **Regla de Fallecimiento**: cada célula con 4 o más vecinos muere por super-población y cada célula con menos de dos vecinos muere por aislamiento.
- **Regla de Nacimiento**: en cada casilla vacía con exactamente **3** células vecinas nace una nueva célula.

Estas tres reglas **se aplican simultáneamente a cada generación** para producir la siguiente. Consideramos que las casillas de la primera y última filas son vecinas, al igual que las de la primera y última columnas, con lo que simularemos una superficie toroidal o "donut". De este modo, todas las casillas tienen exactamente 8 posiciones vecinas (todas las de alrededor, excluida ella misma). A continuación se muestra una población y la generación siguiente que resulta de aplicar las reglas (las células se representan con casillas negras):



Para implementar la simulación, representaremos el estado mediante una matriz de booleanos de dimensión `fils` × `cols`, donde `true` representa célula y `false` casilla vacía. Se pide implementar los siguientes métodos:

- [**1**] `void Inicializa(int fils, int cols, bool [,] mat)`: genera en `mat` una matriz de booleanos de tamaño `fils` × `cols` con las componentes rellenas aleatoriamente con celulas o casillas vacías (con probabilidad 0.5 en ambos casos).
- [**1**] `void Dibuja(bool [,] mat)`: dibuja en pantalla el estado representado en la matriz `mat` utilizando cuadrados blancos para los huecos vacíos y negros para las células. Para obtener una figura proporcionada, cada cuadrado de la matriz se dibujará con dos posiciones horizontales en pantalla (tal como hemos hecho en otros juegos).

Como ya hemos dicho, nuestro juego se desarrolla sobre una superficie toroidal (donut): en la matriz `mat` el borde superior está conectado con el inferior y el izquierdo con el derecho. Es decir, por encima de la fila 0, queda la última fila de matriz; por debajo de la última queda la primera. Y análogo para las columnas de los extremos izquierdo y derecho. Para facilitar el cálculo de células vecinas a una casilla, serán útiles los siguientes métodos [**1**]:

- `int Up(bool [,] mat, int i)`: devuelve el índice de la fila que queda por encima de la fila `i`.
- `int Down(bool [,] mat, int i)`: devuelve el índice de la fila por debajo de la fila `i`.
- `int Left(bool [,] mat, int i)`: devuelve el índice de la columna a la izquierda de la columna `i`.
- `int Right(bool [,] mat, int i)`: devuelve el índice de la columna a la derecha de la columna `i`.

Utilizando estos métodos se simplificará la implementación del siguiente:

- [**1**] `int NumVecinas(bool [,] mat, int x, int y)`: devuelve el número células que hay en las 8 posiciones vecinas a la casilla de coordenadas (`x`, `y`) en la matriz `mat`.

Y a continuación los siguientes:

- [**1**] `bool [,] SiguienteGen(bool [,] mat)`: dada una matriz `mat` que representa una población de células, calcula la generación siguiente aplicando las reglas expuestas al principio del enunciado, y la devuelve una nueva matriz. Para ello utilizará el método anterior `NumVecinas`. Nótese que cada casilla en la nueva población depende únicamente de los vecinos de la población anterior y no de los nacimientos o muertes que se produzcan en la nueva población.
- [**1**] `bool Estable(bool [,] mat1, bool [,] mat2)`: este método servirá para determinar si la población ha alcanzado una configuración estable, es decir, una población que ya no cambia en sucesivas generaciones. Para ello considera dos poblaciones consecutivas representadas en las matrices `mat1` y `mat2`, y comprueba si son iguales (por ejemplo, una población con 4 céluas formando un cuadrado estable).

- [**1**] `void LeeArchivo(string file, bool [,] mat)`: lee del archivo `file` la información de una colonia de células y la devuelve en la matriz `mat`. El archivo tendrá dos líneas iniciales con el número de filas y columnas, y a continuación la matriz propiamente dicha, con 0 para las casillas vacías y 1 para las células. Por ejemplo, para la población inicial de la figura de arriba, el archivo de entrada sería:

```
8
8
00000000
00000000
00010000
00001000
00111000
00000000
00000000
00000000
```

- [**1**] `void EscribeArchivo(string file, bool [,] mat)`: guarda en el archivo `file` el estado de la colonia de células dado en `mat`, en el mismo formato del método anterior.

- [**1**] `void Menu(bool [,] mat)`: preguntará al usuario si quiere generar una población aleatoria inicial o leerla de un archivo. En el primer caso solicitará las dimensiones $fils \times cols$ y utilizará el método `Inicializa`. En el segundo, solicitará el nombre del archivo y utilizará el método `LeeArchivo`.

- [**1**] el método `Main` llamará a `Menu` para obtener la poblacion inicial y luego implementará el bucle principal de la simulación. Este bucle obtiene y dibuja sucesivas generaciones de la población incial hasta obtener una configuración estable (de acuerdo con el método `Estable`) o que el usuario decida terminar pulsado `'q'`. En este último caso se le dará al usuario la opción de guardar la población actual utilizando el método `EscribeArchivo`.

Pista: la población incial del método `LeeArchivo`, tras 32 generaciones vuelve a producir esa misma población.