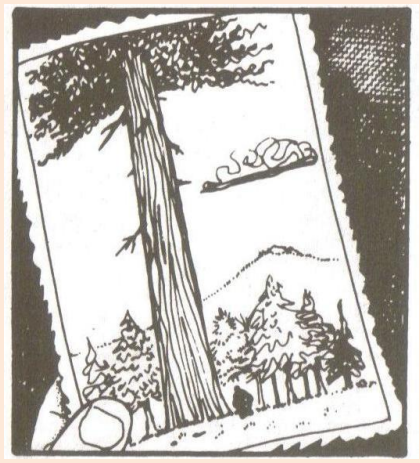


PRÁCTICA FINAL OPTATIVA

DE LA ASIGNATURA

INFORMÁTICA APLICADA

PARA LA TITULACIÓN DE GRADO EN

INGENIERÍA MECÁNICA (UPCT)

CURSO 2013-2014

Implementación del juego de la VIDA

Pedro María Alcover Garau
Pedro José García Laencina
Segunda versión, 30 de diciembre, 2013.

¡No hay como ponerse…!

AUTÓMATAS CELULARES

Un autómata celular es un modelo discreto, estudiado en teoría de la computación, en física y matemáticas, en biología teórica, y en otros ámbitos de la ciencia. Un AUTÓMATA CELULAR consiste en una **REJILLA** regular de **CELDAS** (también llamadas CÉLULAS), cada una de ellas con un número finito de **ESTADOS** posibles (que se describen como un valor en $\mathbb{Z}$). Cada célula puede tomar un valor dentro de un conjunto finito de estados.

La rejilla de un autómata celular puede tener un número finito de dimensiones. Para cada célula existe un conjunto de células dentro de la misma rejilla que llamamos **VECINDARIO**: el vecindario de una célula es el conjunto finito de células en su cercanía. En un autómata es fundamental fijar la estructura regular (topología) en la que sus células se distribuyen: la forma de la rejilla. La topología de la rejilla determina quién forma el vecindario de cada célula.

En un estado inicial (tiempo $t = 0$) se asigna al autómata un estado inicial para cada una de sus celdas. De acuerdo a una colección de reglas predefinidas, en cada nuevo instante (tiempos $t = 1, 2, \ldots$) se define una nueva configuración de la rejilla, donde se decide el estado de cada celda. Se dice que en cada nuevo instante se define una nueva **GENERACIÓN**. La **REGLA** de transformación de un estado al siguiente suele ser una función matemática que determina, para cada celda, el nuevo estado a partir delos estados previos de las celdas de su vecindario. Habitualmente la regla para determinar el nuevo estado de las celdas es la misma para cada una de las celdas de la rejilla y se aplica a todas las celdas de forma simultánea.

El concepto de Autómata Celular fue introducido en los años 40 del siglo XX por Stanislaw Ulam y por John von Neumann. Aunque los autómatas celulares fueron objeto de distintos estudios en los años 1950s y 1960s, fue en la década de los 1970s, con el **JUEGO DE LA VIDA** propuesto por John Conway (un autómata celular de dimensión 2), como los autómatas celulares lograron suscitar el interés más allá del ámbito estrictamente académico. En el año 2002 Stephen Wolfram publicó "***A new Kind of Science***" ("Un nuevo tipo de ciencia") donde mostraba que los autómatas celulares tienen aplicaciones en muchos campos de la ciencia. Actualmente, los autómatas celulares finitos son de interés y gran utilidad en muy diferentes disciplinas.

De forma más formal, podemos definir un autómata celular como una séxtupla (G, G_0, N, Q, f, T), donde:

- $\boldsymbol{G}$ **es la rejilla de células**. Puede ser una rejilla de una (vector), dos (matriz) o más dimensiones.
- $\boldsymbol{G_0}$ **es la asignación inicial de estados** para cada una de las células de la rejilla. La configuración de la rejilla en $t = 0$.
- $\boldsymbol{N}$ **es la** función que asigna a cada célula el conjunto de sus vecinos. La llamamos **función vecindario**.
- $\boldsymbol{Q}$ **es el conjunto de estados posibles**. Como ya ha quedado indicado antes, este conjunto es finito, y las distintas celdas del autómata sólo pueden adoptar un estado de entre los definidos en este conjunto.
- $\boldsymbol{f}$ **es la llamada función de transición**, que asigna en cada nueva iteración del sistema, y para cada célula, un nuevo estado. Para la asignación del nuevo estado se tiene en cuenta el estado de todos los vecinos (vecindario).
- $\boldsymbol{T}$ **es el conjunto de estados finales**.

DESCRIPCIÓN DEL JUEGO DE LA VIDA

EL JUEGO DE LA VIDA

El **JUEGO DE LA VIDA** es un programa de simulación inventado en 1970 por **JOHN HORTON CONWAY**. El juego simula la evolución de una colonia de bichos que vive en un mundo **cuadriculado** cuyas casillas, en un momento dado, pueden estar vacías o habitadas por un solo ser.

El juego de la vida es considerado como un juego de 0 jugadores, lo que significa que la evolución del juego queda enteramente determinada por su estado inicial, y no se requiere en ningún momento la intervención de ningún usuario para el desarrollo del juego. El comportamiento del sistema viene regido por unas reglas que vienen definidas por la llamada función de transición.

Así, para el juego de la vida, el autómata celular queda definido como:

- La rejilla de células G está formada por una matriz bidimensional.
- El estado inicial, G_0, puede ser cualquier combinación de distribución de estados entre las celdas. Nosotros haremos la asignación inicial de estados de forma aleatoria.
- El vecindario N de cada celda está formado por cada una de las ocho celdas que la rodean dentro del reticulado G.
- El conjunto de estados posibles, Q, es bien reducido: o vivo (1) o muerto (0); o celda habitada (1), o celda deshabitada (0).
- Queda pendiente definir la **función de transición**. En el Juego de la Vida esta función f se define de la siguiente manera:
    - NACIMIENTO: Una celda o célula de la rejilla en estado 0 cambia a estado 1 en la siguiente transición si **tres** de los vecinos de la celda están a 1. En cualquier otro caso no hay nacimiento.
    - SUPERVIVENCIA: una célula o celda en estado 1 permanece en ese estado 1 en la siguiente transición si en torno a ella no hay superpoblación (más de 3 vecinos) o si no está completamente aislada (un vecino o ninguno). Es decir, una célula en estado 1 permanece, en la transición, en estado 1 si en torno a ella encuentra **o 2 o 3** vecinos.
    - MUERTE: una celda o célula en estado 1 pasará, en la siguiente transición, al estado 0 si en torno a ella hay superpoblación (**más de 3** células habitadas en torno a ella) o si la célula ha quedado aislada (una o ninguna célula habitada en torno a ella).
- Los posibles estados finales T dependerán de los iniciales. De todas formas, se ha estudiado largamente la distribución de células que llevan a un estado estable donde ninguna de ellas desaparece y ninguna de su alrededor nace. También se han estudiado las configuraciones de patrones que se desplazan en el espacio de la rejilla. Algo diremos en estas páginas.

En los distintos cuadros de la Figura 1 se muestra la evolución desde una distribución inicial aleatoria (a) hasta llegar, mediante sucesivas iteraciones, a un estado final (b) estable, donde no se dan nuevos nacimientos y donde ninguna célula pierde el habitante que tenía en ella. Se ha considerado una rejilla de 10×10 celdas. Cuando el estado de una de estas celdas es 1 (habitada o viva) se ha representado sombreada en color. Además, en cada celda de la rejilla queda indicado el número de celdas habitadas que la rodean.

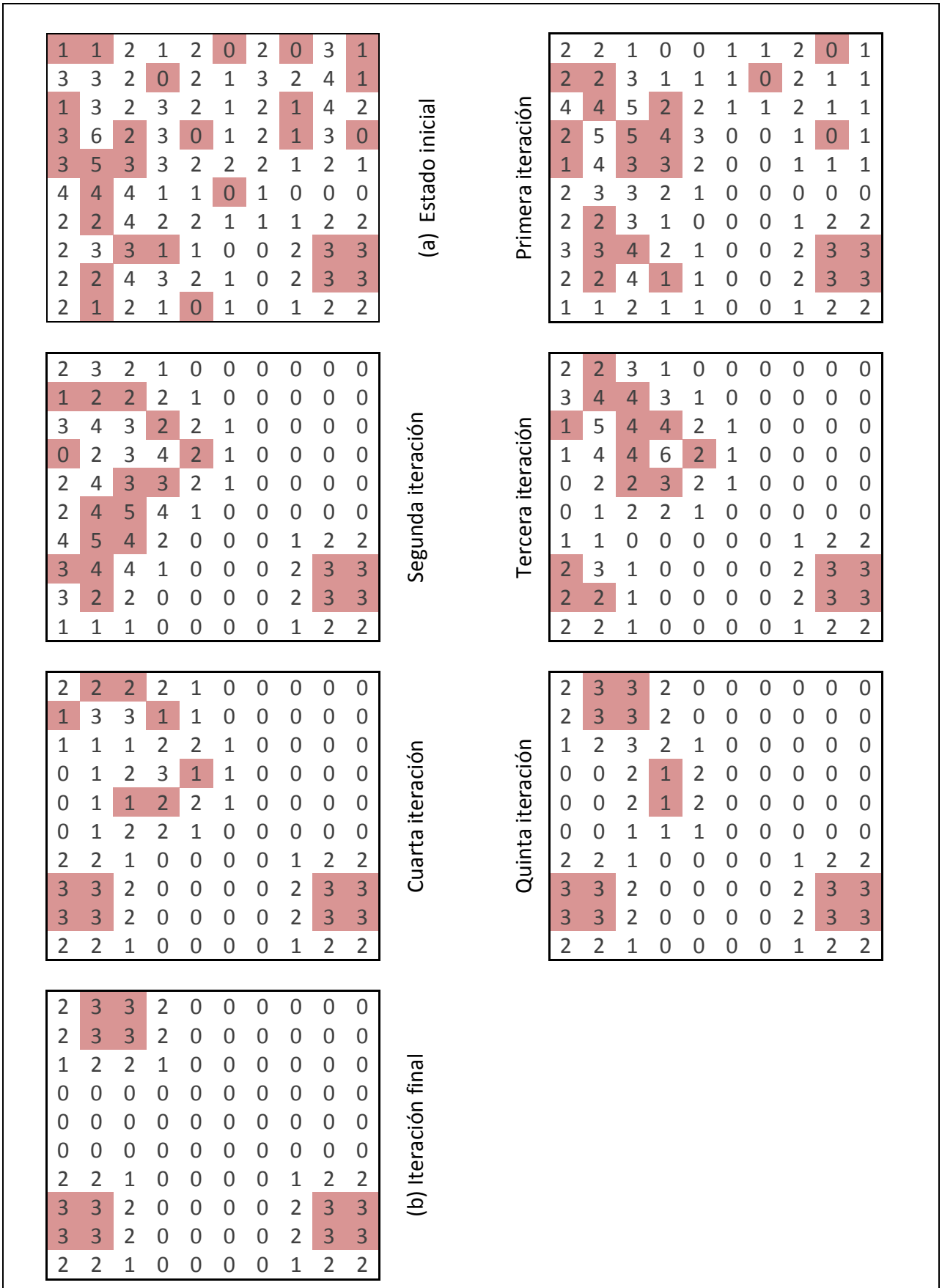


Figura 1. Evolución desde el estado Inicial (a) al final (b) del Juego de la Vida.

Por ejemplo, si analizamos el estado inicial —Figura 1(a)—, la celda de la 2ª fila y la 4ª columna está habitada (se ha representado sombreada) y tiene un cero ya que ninguna de sus celdas vecinas (las ocho celdas que lo rodean) está habitada, es decir, una celda con estado vivo y sin vecindario.

En cada iteración progresa la población de bichos en la rejilla:

- Han muerto los individuos aislados (celdas en estado vivo sin vecindario o con un solo vecino) y también aquellos individuos en zona de superpoblación (celdas con estado vivo y con más de 3 vecinos). Puede ver que todas las celdas sombreadas con un valor de vecindario distinto a 2 o a 3 pasan, en la siguiente iteración, a no estar sombreadas, es decir, a representar una celda deshabitada o una célula muerta.
- Han pasado a estado vivo aquellas celdas muertas que estaban rodeadas de tres células en estado vivo. Puede ver que todas las celdas no sombreadas con un valor de vecindario igual a 3 pasan, en la siguiente iteración, a sí estar sombreadas, es decir, a representar una celda viva o habitada o una célula viva.
- Las celdas donde no había un habitante y con un número de vecinos distinto de 3; y las celdas con habitante con un vecindario igual a 2 ó a 3, no cambian de estado: siguen vivas si vivas estaban; siguen deshabitadas si así lo estaban en la iteración anterior.

Resulta interesante estudiar aquellas configuraciones que definen una estructura constante. Por ejemplo, en la Figura 1, podemos ver en que las cuatro celdas ubicadas entre las filas 8-9 y las columnas 9-10 están siempre habitadas a lo largo de las distintas iteraciones. Como veremos enseguida, hay muchas configuraciones que logran perdurar en el juego de la vida:

(1) Algunas forman los llamados patrones constantes o estáticos, más conocidos como **BLOQUES**; la Figura 2 muestra algunas de ellas. El ejemplo indicado en el párrafo anterior es también un bloque.

(2) También existen configuraciones que configuran patrones llamados **OSCILADORES**, formados por distribuciones que se suceden cíclicamente; se muestran algunos de ellos en la Figura 3.

(3) Por último, existen los patrones que se trasladan por el tablero o rejilla, llamados habitualmente **NAVES ESPACIALES**, y de los que, en la Figura 4, se muestran algunos ejemplos.

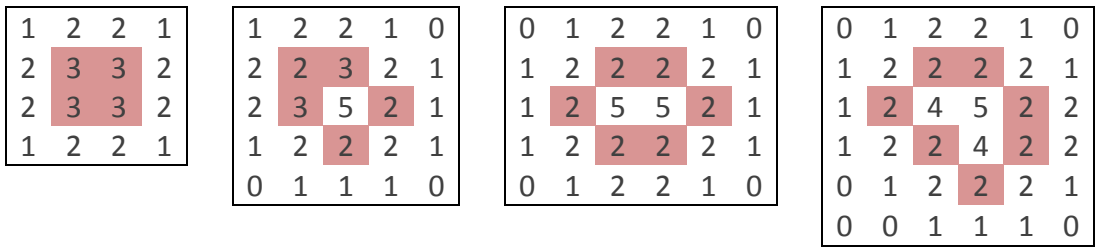


Figura 2. Ejemplos de **BLOQUES**: patrones constantes o estáticos que ni crecen, ni decrecen, ni se mueven. Todas las células en estado vivo tienen 2 ó 3 células vivas a su alrededor, por lo que ninguna muere. Ninguna célula en estado muerto está rodeada de tres células vivas, por lo que ninguna nueva nace. Estos bloques son conocidos como **BLOQUE**, **COLMENA**, **PAN** y **BOTE**.

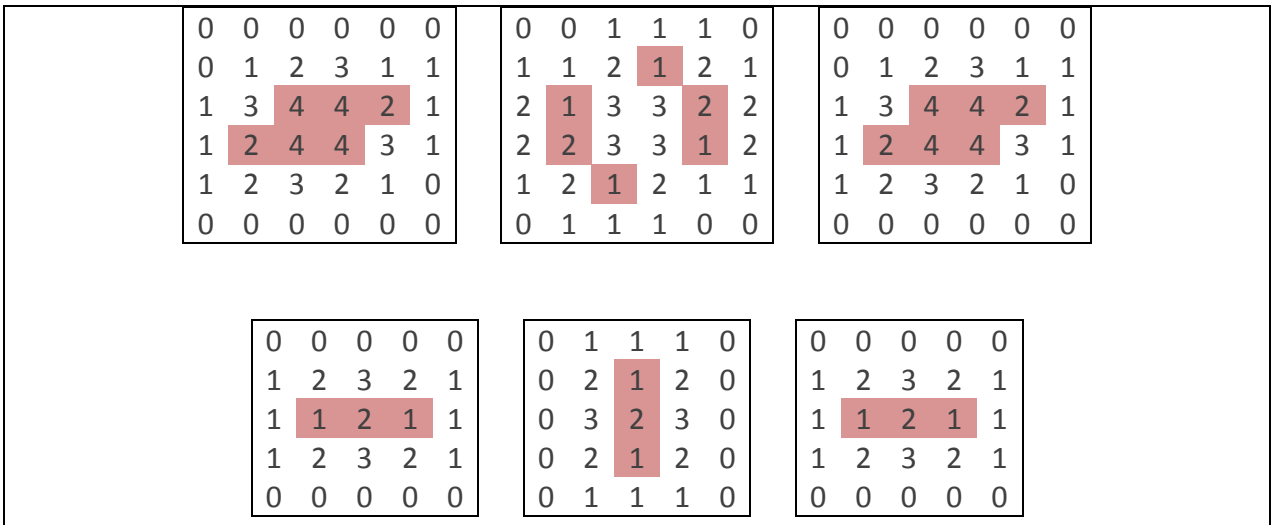


Figura 3. Ejemplos de **OSCILADORES**. En estos dos ejemplos, cada dos iteraciones se vuelve a la misma configuración inicial. El oscilador de arriba es conocido con el nombre de **SAPO** (con periodo 2); el de abajo se le llama **GUIÑO** (también con periodo 2).

El Juego de la Vida ha resultado un sistema que permite, mediante reglas verdaderamente sencillas (la función de transición es realmente simple), observar patrones muy complejos. Es un diseño matemático que ha suscitado un interés sorprendente entre matemáticos y científicos en general. También es interesante en el mundo de la computación. Y aquí nos va a servir para plantear un sencillo desafío de programación.

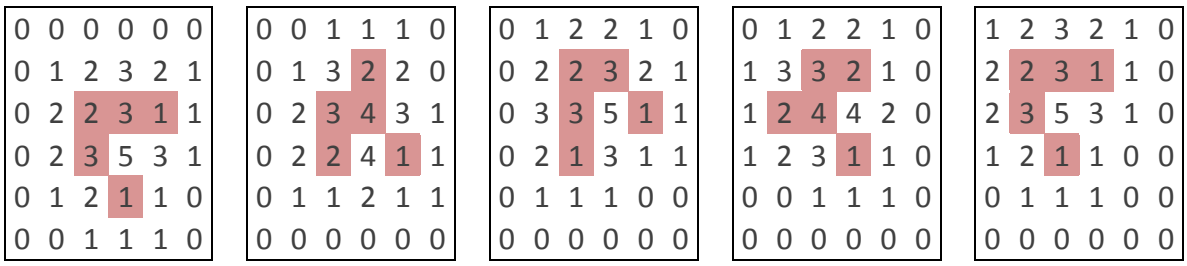
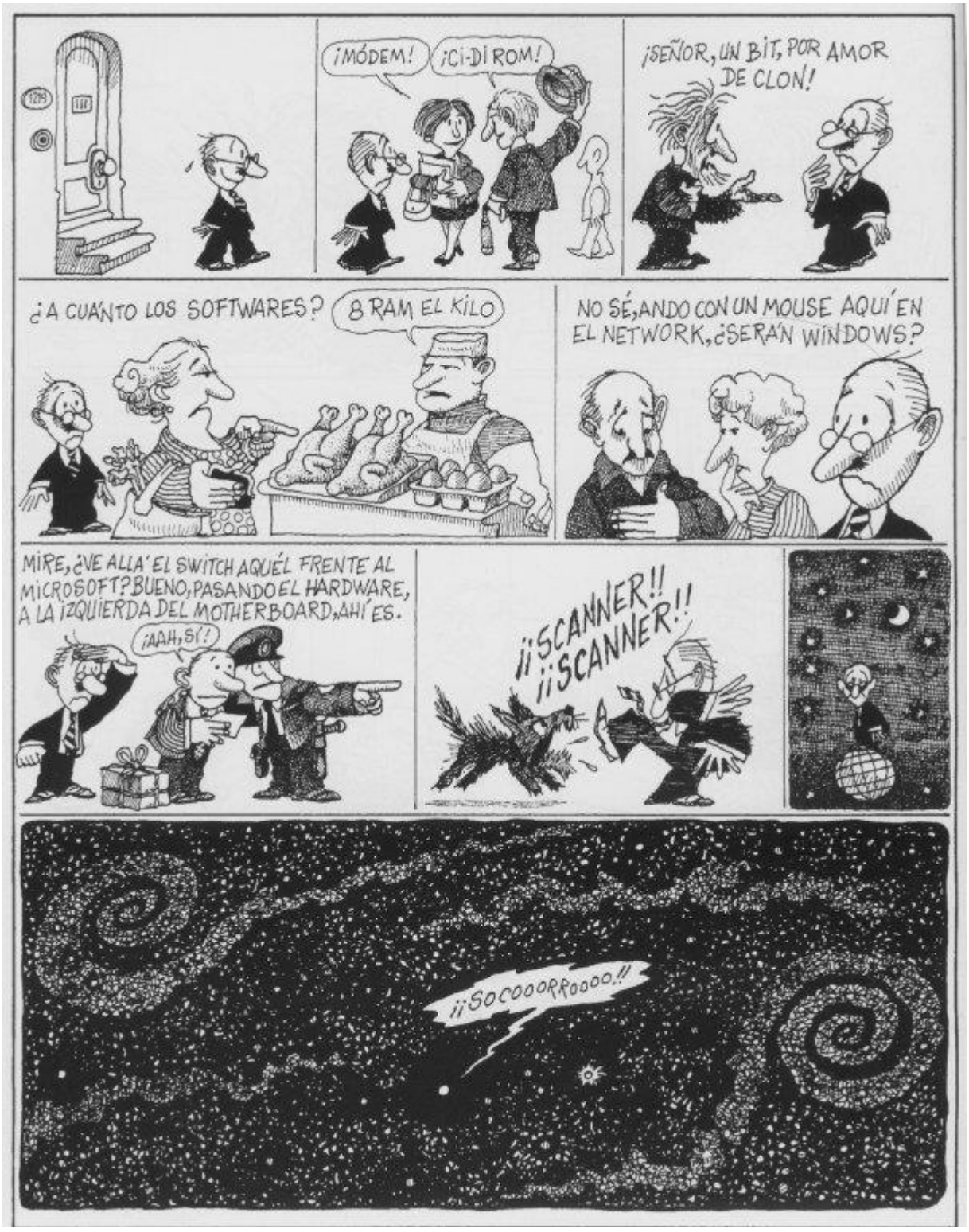


Figura 4: Ejemplo de una **NAVE ESPACIAL**, que en la cuarta iteración vuelve a tomar la misma configuración que tenía en el estado inicial, pero desplazado una posición a la izquierda y una posición arriba. A esta nave espacial se la conoce con el nombre de **PLANEADOR**.



IMPLEMENTACIÓN EN C

¿CÓMO IMPLEMENTAR EL JUEGO DE LA VIDA?

Para la implementación de este juego es necesario primero decidir cómo representaremos la rejilla de células. Y luego, para cada célula, hay que decidir cómo codificamos los dos estados (vivo / muerto) posibles. También debemos decidir la forma en que nuestra rejilla conocerá en todo momento cuántos vecinos tienen cada una de las células o celdas. En definitiva, necesitamos saber qué tipo de dato nos puede servir para codificar, para cada celda, si estado y su vecindario.

Una vez decididos los tipos de dato, la segunda cuestión consiste en decidir qué funciones se van a implementar para lograr una implementación sencilla de la aplicación. Estas funciones habrá entonces que declararlas y que implementarlas. Gracias a una buena definición de tipo de dato para nuestro problema, y gracias a una correcta decisión de qué funciones se deben declarar y definir (implementar), podremos lograr una función `main()` verdaderamente sencilla.

TIPO DE DATO.

Que la rejilla será una matriz de doble dimensión es casi evidente. Viene exigido de la misma naturaleza de la rejilla.

Pero debemos decidir cómo codificamos cada una de las celdas o células. Porque en cada una de ellas hay dos valores importantes que debemos conocer y codificar: el estado (vivo / muerto) y el número de sus vecinos; ambos pueden ser codificados como valores enteros: los dos valores del estado; y los nueve valores posibles (de 0 a 8) del vecindario.

Tenemos varias opciones para codificar estos valores. Elegir uno u otro es una decisión del programador. Estas opciones son:

1. **Crear una estructura de datos** (Ver capítulo 20 del manual de teoría). Es cierto que el desarrollo de la asignatura no llega a este capítulo; pero si se anima a estudiar cómo se crean nuevos tipos de dato verá que es muy sencillo. Se trataría de crear el tipo de dato que se propone en Código 1.

Código 1: Definición de tipo de dato `celda` mediante una estructura.

```
#include <stdint.h>
typedef struct
{
   uint8_t estado;
   uint8_t vecinos;
}celda;
```

Nótese que se propone usar el tipo de dato **`uint8_t`** (enteros sin signo con 8 bits) para codificar el estado y el número de vecinos con la finalidad de optimizar los recursos de memoria necesarios.

Y así, queda creado el tipo de dato llamado `celda`. A partir de este momento podemos trabajar con él, crear variables. Por ejemplo:

```
celda unaCelda;
```

Y para acceder a cada uno de los dos campos de este tipo de dato utilizamos el operador punto (.). Por ejemplo:

```
unaCelda.estado = 0;
unaCelda.vecinos = 3;
```

En el caso anterior, el campo `estado` tiene el valor cero, lo que indica que la celda no está habitada; el campo `vecinos` vale tres, indicando así que tres de sus ocho vecinos están habitados. Sin embargo, quizá resulte más apropiado crear identificadores para los dos estados posibles; por ejemplo, de la manera propuesta en Código 2.

Código 2: Creación de los literales para los dos estados posibles (vivo/muerto) de una celda.

```
#define _VIVO      0xF
#define _MUERTO    0x0
// ...
unaCelda.estado = _MUERTO;
```

Podemos crear también arrays o matrices de este tipo de dato. Así, la rejilla de nuestro juego podría definirse así:

```
#define _SIZE 100
// ...
celda rejilla[_SIZE][_SIZE];
```

Fíjese que con la sentencia anterior se ha definido la rejilla del juego con una matriz cuadrada de un total de 100x100 variables de tipo celda. Y ahora, para hacer referencia a los campos del tipo de dato para cada uno de los elementos de la matriz, diríamos:

```
rejilla[f][c].estado = _VIVO;
rejilla[f][c].vecinos = 0;
```

donde se está indicando que la celda de la fila `f` y la columna `c` tiene un bicho vivo y, además, no tiene ninguna celda vecina habitada. Si aprende a usar este tipo de dato siempre tendrá, en una sola variable, toda la información del sistema de la rejilla.

2. **Utilizar dos matrices para la rejilla** (no recomendada). Si no le convence la solución propuesta en Código 1, entonces puede acudir a ésta que ahora se le propone: una peor solución, pero que no requiere saber cómo se crean nuevos tipos de dato. Usted podría trabajar con las dos siguientes matrices:

```
#include <stdint.h>
#define _SIZE 100
// ...
uint8_t estadoRejilla[_SIZE][ _SIZE];
uint8_t vecinosRejilla[_SIZE][ _SIZE];
```

Y ahora, si usted quiere que en la posición fila `f`, columna `c` el estado de la rejilla sea vivo, y el número de vecinos sea 4 entonces deberá decir:

```
estadoRejilla[f][c] = _VIVO;
vecinosRejilla[f][c] = 4;
```

Con respecto a la primera solución basada en el uso de estructuras, esta segunda solución es peor fundamentalmente por qué ahora cada vez que quiera pasar la rejilla a una función, deberá hacer el paso mediante dos parámetros: uno por cada una de las dos matrices. Siempre podría crear las dos matrices a la vez: `uint8_t rejilla[2][_SIZE][_SIZE];` y así, al indicar a `rejilla[0][f][c]` nos estaríamos refiriendo a la matriz de estados; y al indicar `rejilla[1][f][c]`, a la matriz de vecindario. Así resolveríamos el principal problema señalado.

3. Existe una tercera solución, quizá **la más eficiente de las tres**, que no introduce conceptos que no hayan sido presentados en las clases de teoría de la asignatura, pero que requiere querer ponerse en marcha con los operadores a nivel de bit.

   Se trataría de utilizar cada variable de 8 bits (`uint8_t`) como si estuviera dividida en dos partes. Como el número de vecinos de cada celda no podrá ser nunca mayor de 8, y tampoco será nunca negativo, entonces, en realidad, bastarían con 4 de esos 8 bits que nos ofrece el tipo de dato `uint8_t` para codificar esas valores posibles (desde 0x0 para indicar que no hay vecinos, hasta 0x8 para indicar que hay ocho vecinos: hay valores posibles que nunca se darán). Los otros cuatro bits podríamos usarlos para codificar el estado: en realidad nos bastaría con un bit, pero nos sobran y aquí tomaremos los cuatro.

   La definición del tipo de dato para una celda podría ser, entonces, el recogido en Código 3.

Código 3: Definición de tipo de dato `celda` como campo de bits.

```
#include <stdint.h>
// ...
typedef uint8_t celda;
```

Y en una variable de este tipo codificamos la información sobre el estado (habitado o vacío; vivo o muerto) y sobre el número de vecinos (un valor entre 0 y 8). Podemos tomar los cuatro bits más significativos de una variable de tipo de dato `celda` como el lugar donde se codifica el estado de la celda o célula: por ejemplo `0xF` o `_VIVO` (para indicar habitado), o `0x0` o `_MUERTO` (para indicar celda vacía). Y podemos tomar los cuatro bits menos significativos de esa variable como el lugar donde se guarda el número de vecinos que tiene la celda. Así, por ejemplo, si la variable de tipo celda tiene el valor `0xF2`, indica que la celda

está habitada y que tiene 2 vecinos. Si vale `0x03`, entonces la celda está vacía y dos de sus ocho celdas colindantes están habitadas.

Así, los valores posibles son desde `0x00` hasta `0x08` (9 valores para una celda deshabitada) y desde `0xF0` hasta `0xF8` (otros 9 valores posibles para una celda habitada).

Con esta representación de los datos, *la matriz de información ocupa menos memoria que en las dos soluciones anteriores propuestas*. Ahora el problema está en la asignación de los valores y a la lectura de la información. Esto se resuelve de manera sencilla si definimos una colección de funciones *ad hoc*, como las que se declaran y definen en Código 4. Estas funciones también podrían definirse si se eligiera la definición del tipo de dato sugerido en Código 1. En Código 5 se muestra cómo sería esa implementación en ese caso. No presentamos la colección de estas cuatro funciones para el caso en que el programador haya elegido consignar la información de cada celda usando doble matriz.

Código 4: Declaración y definición de funciones que asignan y devuelven los valores de estado y de vecindad de cualquier expresión del tipo celda definido en Código 3.

```
void    setEstado       (celda *c , uint8_t estado);
void    setVecindad     (celda *c , uint8_t vecindad);
uint8_t getEstado       (celda c);
uint8_t getVecindad     (celda c);

void setEstado      (celda *c , uint8_t estado)
     {   *c = estado ? (*c | 0xF0) : (*c & 0x0F);            }
void setVecindad    (celda *c , uint8_t vecindad)
     {   *c = (0xF0 & *c) ^ (0x0F & vecindad);               }
uint8_t getEstado   (celda c)      { return (c & 0xF0) >> 4;  }
uint8_t getVecindad (celda c)      { return (c & 0x0F);       }
```

Código 5: Declaración y definición de funciones que asignan y devuelven los valores de estado y de vecindad de cualquier expresión del tipo celda definido en Código 1. Se han omitido los prototipos de estas funciones ya que coinciden con los prototipos de Código 4.

```
void setEstado      (celda *c , uint8_t estado)
     {   c->estado = estado ? _VIVO : _MUERTO ;              }
void setVecindad    (celda *c , uint8_t vecindad)
     {   c->vecinos = vecindad;                              }
uint8_t getEstado   (celda c)        {   return c.estado;     }
uint8_t getVecindad (celda c)      {   return c.vecinos;      }
```

Cada programador puede implementar su propio código. Y cada uno puede definir las funciones que desee. Aquí se presenta (ver Código 6) una posible función principal, que invoca a su vez a una serie de funciones que se explicarán en la siguiente subsección de esta memoria. Cuál sea el tipo de dato que al final se hay escogido (opción 1 ó 3 del epígrafe anterior) no afecta a los prototipos de las funciones que se van a presentar. Quien decida trabajar con el modelo de las dos matrices deberá declarar y definir sus funciones con un parámetro más: el de la segunda matriz. Pero esa solución no la presentamos en esta memoria.

Código 6: Posible función main.

```
#define _SIZE 20
#define _DENSIDAD 30

int main(void)
{
	celda mundo[_SIZE][_SIZE];
	uint16_t bichos;

// Inicializamos la rejilla con la función inicializarMundo.
// Se creará una rejilla con una densidad de células con vida
// indicada en el tercer parámetro. Esta función devuelve como
// valor el número de celdas en estado de vida.

	bichos = inicializarMundo(_SIZE, mundo, _DENSIDAD);

// Mientras queden celdas en estado de vida...
	while(bichos)
	{
		system("cls");
		contarVecinos(_SIZE, mundo);
		mostrarMundo(_SIZE, mundo);
		bichos = supervivencia(_SIZE, mundo);
		usleep(150000);
	}
	return 0;
}
```

La llamada a la función system(), para cuyo uso se debe incluir el archivo de cabecera stdlib.h, se emplea para lanzar a ejecución, desde el programa, una orden del sistema operativo. Esa orden que se desea lanzar en el momento de la ejecución de la función system() debe ir escrita entre comillas dobles. La orden "cls" del sistema operativo Windows borra toda la ventana de ejecución de comandos y deja el cursor en la esquina

superior izquierda: es como un reinicio de la pantalla de ejecución. Si trabaja en una máquina Linux la orden deberá ser `"clear"`.

La función `usleep()` interrumpe el hilo de ejecución durante tantos microsegundos como indique el valor recibido entre paréntesis. En la función `main` sugerida se indican 150000 microsegundos, lo que implica que el proceso iterado con el **`while`** se podrá ejecutar hasta 6 veces por segundo. Para hacer uso de esta función hay que incluir el archivo de cabecera `unistd.h`.

El programa se ejecutará continuamente mientras en nuestra rejilla quede alguna celda con el estado a `_VIVO`. Esa información se calcula tanto en la función `inicializarMundo()` como en la función `supervivencia()`, que determina una y otra vez el nuevo estado de la rejilla a partir de las vecindades y vidas de su estado anterior. Pero eso se explica en la siguiente subsección.

TODAS LAS DEMÁS FUNCIONES

Con la solución aquí conducida y sugerida (pero que no quedará mostrada en su totalidad), es necesario declarar las funciones que se muestran en Código 7.

Código 7: Declaración de las posibles funciones a implementar en nuestra aplicación. Válido para aquellos que hayan elegido crear el tipo de dato `celda`, en la forma indicada en Código 1 o en Código 3.

```
void mostrarMundo(uint16_t d , celda mundo[d][d]);
uint16_t inicializarMundo(uint16_t d, celda mundo[d][d], uint16_t p);
void contarVecinos(uint16_t d , celda mundo[d][d]);
uint16_t supervivencia(uint16_t d , celda mundo[d][d]);
```

La primera función incluida en Código 7, `mostrarMundo()`, se encarga de imprimir por pantalla, en una forma tal como se muestra en la Figura 1, el estado de la rejilla después de cada nueva iteración. Una posible definición de esta función se le ofrece en Código 8.

En Código 8 puede verse cómo se ha intentado evitar llamar repetidamente a la función `printf()`. Esa función es lenta en su ejecución, y cuando se trata de mostrar una y otra vez, a razón de 6 veces por segundo toda la matriz rejilla del juego de la vida, esas llamadas retrasan, de hecho, la ejecución del programa. Además, al intentar mostrar la rejilla mediante múltiples llamadas a la función `printf()`, el efecto visual que se produce es el de un constante parpadeo, incómodo a la vista. Es mejor construir una cadena de texto con todos los valores de la matriz que se debe enseñar. Eso es lo que se hace sobre la cadena `matrizMundo()`, gracias a las funciones de `stdio.h` `sprintf()` y de `string.h` `strcat()`.

Código 8: Posible implementación de la función `mostrarMundo()`. Este código presupone el tipo de dato `celda` en una de las formas que se muestran en Código 1 o 3.

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#define _BICHO       2
#define _NOBICHO     '.'
/*
   El propósito de esta función es crear una cadena de texto, para
   mostrar por pantalla mediante la función printf(), que muestre
   el cuadro que representa el estado de la rejilla del juego de la
   vida. Esa cadena de texto se construye mediante la función
   sprintf(): puede verse documentación sobre esta función en la web.
*/

void mostrarMundo(uint16_t d , celda mundo[d][d])
{
     uint16_t f, c;
     char matrizMundo[(2 * d * d) + d + 1];
     char sibicho[4], noBicho[4];
     sprintf(siBicho , " %c" , _BICHO)
     sprintf(noBicho , " %c" , _NOBICHO)
/*
   * Las variables f y c se emplean para recorrer la matriz mundo.
   * En el array matrizMundo se crea la cadena de texto que se ha de
   imprimir en cada nueva iteración del sistema. Es mejor crear la
   cadena de texto con todo lo que se debe imprimir, y hacer una
   sola llamada a la función printf, que ir mostrando la rejilla
   a base de sucesivas y reiteradas llamadas a la función printf.
   * El array aux se emplea como cadena auxiliar para construir la
   cadena matrizMundo.
*/
     matrizMundo[0] = 0; // Queda inicializada la cadena de texto
     for(f = 0 ; f < d ; f++)
     {
          for(c = 0 ; c < d ; c++)
          {
               strcat(matrizMundo,
                    getEstado(mundo[f][c]) ? siBicho : noBicho);
          }
          strcat(matrizMundo, "\n");
     } // fin de la iteración for

// Queda simplemente mostrar por pantalla la cadena matrizMundo.
     printf("%s\n\n", matrizMundo);
     return;
```

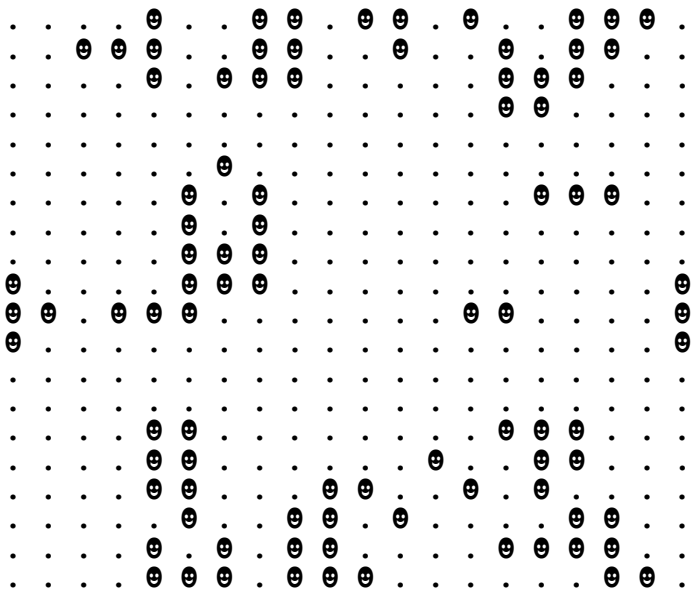


Figura 1: Ejemplo de cómo muestra, la función `mostrarMundo()`, una posible iteración del juego de la vida. Está creado sobre un tamaño de 20 X 20.

La siguiente función incluida en Código 7 es `inicializarMundo()` que se encarga de asignar a la rejilla del juego de la vida los valores iniciales. Ésta es la primera función invocada en esta implementación de la función `main()`. Una posible implementación de esta función se recoge en Código 9. No se explica en esta memoria el uso de las funciones `srand()`,`rand()` (de `stdlib.h`), y `time()` (de `time.h`): el alumno puede encontrar una descripción en el manual de prácticas, en el anexo I recogido al final de la práctica n. 6. La función sugerida en Código 9 asigna un porcentaje de celdas con vida igual al indicado como tercer parámetro.

Código 9: Posible implementación de la función `inicializarMundo()`.Este código presupone el tipo de dato `celda` en una de las formas que se muestran en Código 1 o 3.

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>

#define _VIVO   0xF
#define _MUERTO 0x0
/*
   Esta función asigna valores iniciales a la rejilla del juego de
   la vida. Un porcentaje de celdas con vida aproximadamente igual
   al indicado en el tercer parámetro de la función. La rejilla
   se pasa a la función en el segundo parámetro. La dimensión de
   la rejilla se recibe en el primer parámetro.
*/
```

Código 9: (Continuación).

```
uint16_t inicializarMundo
(uint16_t d , celda mundo[d][d] , uint16_t porcentaje)
{
	int16_t f, c;
	int16_t contadorBichos;

	srand(time(NULL));

	if(porcentaje < 0 || porcentaje > 100) return 0;

	for(contadorBichos = 0 , f = 0 ; f < d ; f++ , srand(rand()))
	{
		for(c = 0 ; c < d ; c++)
		{
			if(rand() % 101 < porcentaje)
			{
				setEstado(&mundo[f][c] , _VIVO);
				contadorBichos++;
			}
			else setEstado(&mundo[f][c] , _MUERTO);
		}
	}
	return contadorBichos;
}
```

No se presenta en esta memoria una implementación concreta de las otras dos restantes funciones sugeridas en Código 7: `contarVecinos()` y `supervivencia()`. El alumno deberá hacer su propia implementación de estas funciones si quiere que la aplicación (en su versión más estándar, sin incluir mejora alguna) pueda ser ejecutada y muestre una primera versión del Juego de la Vida. Como verá a lo largo de las páginas que siguen en esta memoria, el alumno puede implementar, además, distintas mejoras a la aplicación, que la hagan posiblemente mejor y, desde luego, más completa. Desde luego, la nota final podrá ser mejor cuantas más y mejores sean las mejoras introducidas a esta descripción inicial aquí presentada. En el siguiente apartado se dan algunas ideas al respecto.

ALGUNAS VARIACIONES PARA IMPLEMENTAR

El juego de la vida admite bastantes variaciones. Una vez se ha implementado el juego tal y como se solicita en las páginas anteriores de esta memoria, se pueden incorporar muchas modificaciones. Algunas de esas nuevas implementaciones incluyen variaciones de contorno. Otras suponen mejoras en el desarrollo del juego, o en la oferta que se ofrece al usuario. En este apartado se sugieren mejoras posibles. El alumno que aspire a una buena nota por la solución presentada para esta práctica deberá incluir, en su implementación, algunas de esas mejoras.

Variaciones de contorno.

Se puede considerar que el mundo representado por la rejilla tiene estructura toroidal, esto es, que sus lados inferior y superior están conectados, siendo vecinas las casillas de la primera y última filas, y lo mismo con las de sus lados izquierdo y derecho.

Variaciones en las condiciones.

El Juego de la Vida que se ha propuesto en estas páginas, el llamado Juego Estándar, es en el que nace una célula si en la celda correspondiente hay tres celdas vecinas vivas, y una célula sigue viva si tiene 2 ó 3 células vecinas vivas, y muere en otro caso. Esta configuración se simboliza como “23/3”: el primer número o lista de números es lo que requiere una célula para que siga viva, y el segundo es el requisito para su nacimiento.

Así [copio de Wikipedia], “16/6” significa que una célula nace si tiene 6 vecinas, y vive siempre que haya 1 ó 6 vecinas. Una configuración parecida a la estándar, y que se conoce como HighLife, es la definida como “23/36”: nace una nueva célula si hay 3 ó si hay 6 vecinos; una célula sigue viva si la rodean 2 ó 3 vecinos.

Se pueden considerar, así, bastantes variaciones al juego de la vida estándar. Muchas de esas variaciones crearán un mundo desolado, donde casi cualquier población tienda rápidamente a la muerte de todos sus individuos; otras configuraciones, en cambio, podrán llevar a mundos donde casi de forma inmediata se alcanza la superpoblación.

Puede introducir una variación en el código de su práctica, de forma que el usuario indique, al inicio del juego, con qué configuración debe evolucionar la distribución inicial aleatoria de celdas vivas de su rejilla. Si desea realizar esta modificación para la aplicación que presente a evaluación, será conveniente que tenga en mente este propósito cuando implemente la función `supervivencia()`.

Variaciones en la creación de nuevos estados iniciales.

Hasta el momento hemos supuesto que el estado inicial, a partir del cual podemos ver la evolución de estados de nuestra rejilla, se crea de forma aleatoria. Pero quizá sería útil, o conveniente, poder ingresar los estados iniciales a partir de un archivo de texto. Para ello debería conocer cómo se trabaja con archivos de disco. No es complicado, y tiene información suficiente para comprender cómo se hace en el Capítulo 21 del manual de teoría. Y, desde

luego, puede acudir a tutorías si desea comprender cómo se realiza y quiere, luego, intentar hacer una modificación en esta práctica final. Podría así, usted, cargar algunas configuraciones iniciales del juego de la vida que son, desde luego, dignas de ver. Puede hacer búsquedas de ello en internet, y podrá comprobar que hay configuraciones iniciales que evolucionan de forma sorprendente.

Mejoras sobre el código sugerido.

Como podrá comprobar cuando termine la implementación de la práctica y logre que el programa funcione, la función `main()` determina que el juego finalice una vez que se llega a un estado en el que todas las células están en estado `_MUERTO`. Eso es correcto, desde luego, pero la realidad del juego es que casi nunca se da ese caso: lo más frecuente es que el juego termine en una situación donde todas las celdas en estado `_VIVO` tienen una configuración tal que o se comportan como Bloques (ver Figura 2), o se comportan como Osciladores (ver Figura 3). Se podría hacer alguna variación en alguna de las funciones (o de la función `main()`) de manera que se pudiera detectar el final del juego, o bien cuando se llega a una situación en la que dos iteraciones consecutivas describen idéntica rejilla, o bien cuando dos iteraciones consecutivas, separadas por una iteración intermedia, resultasen idénticas. Es verdad que hay osciladores de periodo mayor que 2, pero quizá podamos no considerar esas excepciones poco habituales.

ARCHIVOS DE CÓDIGO

Le recomendamos que, dentro del proyecto que se va a crear y donde se recogerá todo el código de la aplicación, cree 3 archivos:

- **`funciones.h`**: donde se recogerán todas las definiciones de tipo de dato, y todas las directivas `define`. También deberán recogerse todas las declaraciones de todas las funciones.
- **`funciones.c`**: donde se recogen todas las implementaciones de las funciones declaradas en `funciones.h`. Este archivo deberá tener, al inicio, la siguiente directiva:

  ```
  #include "funciones.h"
  ```

  Y también aquellas inclusiones de archivos de cabecera que se necesiten para el uso de las funciones en ese archivo.
- **`main.c`**: donde se recoge la función principal. Este archivo también deberá recoger la inclusión del archivo de cabecera `funciones.h` , de la misma manera que ha sido incluida en `funciones.c`.

INDICACIONES FINALES

Se le ha facilitado ayuda: se le ha explicado qué debe hacer el código que usted va a implementar (si quiere). Se le ha facilitado parte de este código. Pero no tiene todo el trabajo hecho. Deberá comprender, si de verdad quiere emplearlo, el código que se le facilita. Deberá ser capaz de ensamblarlo, y lograr montar felizmente una aplicación completa que ofrezca al usuario la posibilidad de jugar al juego de la vida. Insisto: si no comprenden el código que se les facilita, no podrán saber cómo ensamblarlo.

Desde luego, puede afrontar el trabajo propuesto sin hacer uso alguno de lo que en esta guía se les dice. Siempre hay alumnos que así lo hacen, y siempre logro ver, gracias a ellos, soluciones originales (a mí me lo parecían).

No se desanimen ante el enunciado de la práctica. Que sea largo no es un argumento contra él, sino más bien a favor: si fuera más largo, quizá sería porque aún les facilitaba más código. Pero no puedo darles más, porque poco es lo que falta para tenerlo terminado.

No deje este trabajo para “mas tarde”. No decida que es demasiado difícil para usted. Fundamentalmente porque no es cierto.

¡Ánimo!

(Continúa en la página siguiente…)


¡LA MUY BESTIA!
¡YO!...¡YO EL
PRIJÁ-JÁ-JÁ-JÁ
DEL GRAJÁ-JÁ-JÁ-J


SALUD, MANOLITO ¿POR QUÉ TAN CABIZBAJO?
¿TE ACORDÁS QUE YO DECÍA QUE LO BUENO DE LOS PRIMEROS DÍAS DE CLASE ES QUE LA MAESTRA TODAVÍA NO CONOCÍA BIEN A SUS ALUMNOS?
SÍ
¿Y QUE PARA ELLA POR AHORA ÉRAMOS TODOS IGUALES, SIN MEJORES, NI PEORES, NI NADA?
SÍ
BUENO, HOY ME CORRIGIÓ EL PRIMER DICTADO.... ¡Y SE ACABÓ LA DEMOCRACIA!