TP 4B : Implémentation d'un automate cellulaire simple
Le jeu de la vie

Coline Trehout

7 avril 2021

Table des matières

Liste des tableaux

Table des figures

1 Introduction

L'objectif de ce TP est d'implémenter le jeu de la vie qui est un automate cellulaire simple. Les règles du jeu de la vie ont été définies par le mathématicien John Conway en 1970. Malgré des règles simples et déterministes, l'évolution macroscopique du système peut devenir complexe et imprévisible. C'est pourquoi cet automate continue de fasciner mathématiciens et informaticiens plus de 50 ans après sa création. Le principe du jeu de la vie ainsi que son implémentation en langage C seront expliqués puis plusieurs observations intéressantes seront présentées.

2 Principe du jeu de la vie

L'automate cellulaire du jeu de la vie fonctionne sur une matrice en 2 dimensions. Chaque case contient une cellule qui est dans un état binaire, elle est soit vivante soit morte. On applique le voisinage de Moore : chaque cellule est donc entourée de 8 voisins. Sur la figure 1, la cellule noire est vivante et ses voisins sont représentés en vert.

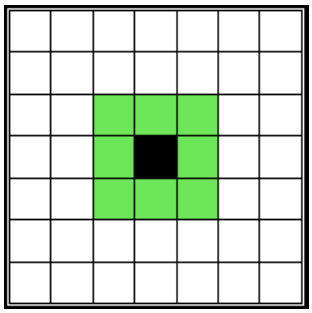

FIGURE 1 – Représentation du voisinage de Moore de la cellule noire [1]

L'état de l'automate au temps t est uniquement fonction de son état au temps t-1. Une fois la configuration initiale choisie, le système évolue selon les règles suivantes :

— Une cellule morte entourée de 3 voisins naît
— Une cellule vivante avec 0 ou 1 voisin meurt d'isolement
— Une cellule vivante avec au moins 4 voisins meurt de surpopulation
— Une cellule vivante avec 2 ou 3 voisins survit

C'est l'analogie entre ces règles et certains critères d'évolution de populations de bactéries qui a conduit à donner à cet automate le nom de jeu de

la vie.

Le système peut évoluer sur une simple grille carrée ou alors dans un univers torique (voir figure 2). Ainsi le voisinage est étendu pour les cellules situées en bordure de la grille. Cela permet notamment à certains motifs de pouvoir se propager à l'infini.

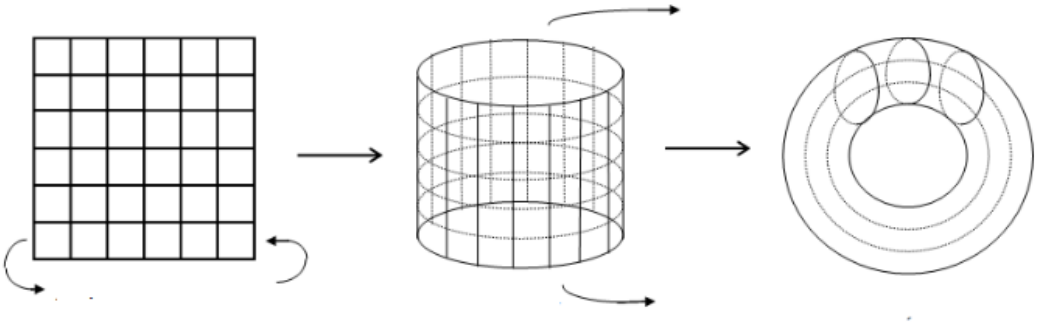

FIGURE 2 – Représentation d'un univers torique

3 Implémentation en langage C

L'implémentation du jeu de la vie a été faite en langage C. Le code complet est disponible en annexe 1. Dans cette implémentation, une cellule morte est représentée par un point tandis qu'une cellule vivante est représentée par le caractère 'X'.

3.1 Initialisation de la grille

Pour implémenter la grille de jeu, nous avons besoin de 2 tableaux de caractères à 2 dimensions. Le premier tableau correspond au système à l'instant t et le second à l'instant t+1. Les 2 tableaux dynamiques `grille1` et `grille2` sont déclarés dans le main. Une fois la taille de la grille souhaitée saisie par l'utilisateur, ils sont initialisés grâce à la fonction `creationGrille`. Par défaut ils sont remplis de cases mortes.

Afin de proposer une certaine modularité à l'utilisateur, différents choix lui sont proposés au démarrage du jeu. Il peut tout d'abord choisir le type d'univers souhaité, c'est-à-dire torique ou non. Ensuite, il peut choisir la taille de la grille parmi 2 possibles : 10^2 ou 50^2. Puis il peut choisir la configuration

initiale de la grille. Par exemple, il peut choisir d'afficher un glider ou encore une initialisation aléatoire avec choix du pourcentage de cellules vivantes au départ. Ces choix se font grâce à des menus `switch` associés à la commande de saisie `scanf`. Enfin, l'utilisateur peut saisir le nombre d'itérations qu'il souhaite observer. Un exemple de menu pour le choix de la taille de la grille est présenté ci-dessous.

```
1  printf("Veuillez choisir la taille de la grille de jeu :\n\
     n");
2  printf("1 : grille de taille 10*10 \n");
3  printf("2 : grille de taille 50*50 \n");
4  printf("\nVotre choix : ");
5
6  scanf("%d", &choix);
7
8  // choix de la taille de la grille
9  switch (choix)
10 {
11   case 1 :
12     tailleGrille = 10;
13   break;
14
15   case 2 :
16     tailleGrille = 50;
17   break;
18
19   default :
20     printf ("Erreur de saisie\n");
21     exit (0);
22   break;
23 }
```

3.2 Déroulement du jeu

Une fois toutes les informations saisies, la grille de départ est affichée grâce à la fonction `afficheGrille` pendant une seconde pour que l'utilisateur ait le temps d'observer la situation initiale.

Ensuite, la fonction `jeu` est appelée dans une boucle `for` pour lancer le jeu. Dans cette boucle, 2 appels disctincts sont possibles suivant la parité de i. Si i est pair, `grille1` sera considéré comme la grille à l'instant t et `grille2` sera modifiée pour correspondre à l'instant t+1. Dans le cas où i est impair, c'est le contraire. Les tableaux vont donc être modifiés tour à tour. Cela permet de ne pas avoir à recopier le tableau `grille2` dans `grille1` à

chaque itération. La fonction `jeu` est la même que l'on soit dans un univers torique ou non. En effet, seule la manière de compter le nombre de voisins vivants est différente.

L'instruction `usleep (300000)` permet de mettre en pause le programme pendant 0.3 secondes afin de pouvoir observer l'évolution du système à chaque instant. Cette pause n'est faite que si l'entier `debug` est égal à 1. C'est le cas par défaut mais l'utilisateur peut modifier cette valeur s'il souhaite seulement observer la configuration finale.

Lorsque le jeu est terminé, la mémoire allouée pour les tableaux `grille1` et `grille2` est libérée.

Un extrait de la fonction `main` est présenté ci-dessous ainsi que la fonction `jeu`.

```
1   printf ("\nDébut du jeu :\n");
2
3   afficheGrille (grille1, tailleGrille);
4
5   // pause de 1 seconde pour voir l'état initial
6   usleep (1000000);
7
8   for (i = 0; i < iter; i ++)
9   {
10    // alternance des grilles à t et t+1
11    if ( i % 2 == 0)
12    {
13      jeu (grille1, grille2, tailleGrille, tore);
14      afficheGrille (grille2, tailleGrille);
15    }
16    else
17    {
18      jeu (grille2, grille1, tailleGrille, tore);
19      afficheGrille (grille1, tailleGrille);
20    }
21
22    // pause de 0.3 secondes en mode debug
23    if (debug == 1)
24    {
25      usleep (300000);
26    }
27  }
28
29  printf ("Fin du jeu\n");
30
31
```

```
32    // libération mémoire
33    libereGrille (grille1, tailleGrille);
34    libereGrille (grille2, tailleGrille);
35
36    return 0;
```

```
1  void jeu (char ** grille1, char ** grille2, int taille, int
       tore)
2  {
3    int i, j, nv = 0;
4
5    for (i = 0; i < taille ; i ++)
6    {
7      for (j = 0; j < taille; j ++)
8      {
9        // univers non torique
10       if (tore == 0)
11       {
12         nv = nbVoisins (grille1, taille, i, j);
13       }
14       // univers torique
15       else
16       {
17         nv = nbVoisinsTore (grille1, taille, i, j);
18       }
19
20       // la cellule vivante survit si elle a 2 ou 3 voisins
21       if ( grille1 [i][j] == 'X' && (nv == 2 || nv == 3) )
22       {
23         grille2 [i][j] = 'X';
24       }
25       // la cellule morte nait si elle a 3 voisins
26       else if ( grille1 [i][j] == '.' &&  nv == 3 )
27       {
28         grille2 [i][j] = 'X';
29       }
30       // la cellule meurt ou reste morte dans les autre cas
31       else
32       {
33         grille2 [i][j] = '.';
34       }
35     }
36   }
37 }
```

3.3 Implémentation dans un univers non torique

Pour un univers non torique, la fonction `jeu` fait appel à la fonction `nbVoisins` qui lui renvoie le nombre de voisins de la case étudiée. Cette fonction contient une double boucle permettant de parcourir le carré de taille $3*3$ entourant la cellule considérée (voir figure 1). `nbVoisins` fait appel à la fonction `indiceValide` qui renvoie 1 si l'indice de la case et valide et 0 sinon. L'indice est valide s'il est différent de celui de la cellule étudiée (une cellule ne pouvant pas être son propre voisin) et s'il ne dépasse pas les dimensions de la grille. Si l'indice de la case est valide et que la cellule est vivante, le compteur `nv` est incrémenté. Enfin, lorsque le parcours du carré de taille 3 est terminé, le nombre de voisins est renvoyé à la fonction `jeu`.

```
int nbVoisins (char ** grille, int n, int i, int j)
{
  int nv = 0, k, l, test = 0 ;

  for (k = (i - 1); k <= (i + 1); k ++)
  {
    for (l = (j - 1); l <= (j + 1); l ++)
    {
      test = indiceValide (grille, n, i, j, k , l);

      // si l'indice du voisin est valide et si le voisin est
    vivant
      if ( test  == 1 && grille [k][l] == 'X' )
      {
        nv ++;
      }
    }
  }

  return nv;
}
```

```
int indiceValide (char ** grille, int n, int i, int j, int k,
    int l)
{
  // cellule étudiée
  if ( k == i && l == j )
  {
    return 0;
  }
  // cellule en dehors de la grille
  else if ( k < 0 || l < 0 || k >= n || l >= n )
  {
    return 0;
  }
```

```
13    else
14    {
15      return 1;
16    }
17  }
```

3.4 Implémentation dans un univers torique

Dans un univers torique, le voisinage des cellules en bordure de grille est étendu. Cela permet notamment à un glider de continuer sa progression sans jamais s'arrêter. Dans ce cas c'est la fonction `nbVoisinsTore` qui compte le nombre de voisins de la cellule étudiée et qui le renvoie à la fonction `jeu`. On utilise le modulo et l'addition pour traiter les cellules qui seraient normalement en dehors de la grille.

```
1  int nbVoisinsTore (char ** grille, int n, int i, int j)
2  {
3    int nv = 0, k, l, kv, lv;
4
5    for (k = (i - 1); k <= (i + 1); k ++)
6    {
7      for (l = (j - 1); l <= (j + 1); l ++)
8      {
9        kv = k;
10       lv = l;
11
12       // adaptation de l'indice dans l'univers torique
13       if ( k < 0 )
14       {
15         kv = k + n;
16       }
17
18       if ( l < 0 )
19       {
20         lv = l + n;
21       }
22
23       if ( k >= n )
24       {
25         kv = k % n;
26       }
27
28       if ( l >= n )
29       {
30         lv = l % n;
31       }
32
```

```
33        // si l'indice du voisin n'est pas la cellule actuelle
      et si le voisin est vivant
34        if ( ( kv != i || lv != j ) && grille [kv][lv] == 'X' )
35        {
36          nv ++;
37        }
38      }
39    }
40
41    return nv;
42  }
```

4 Observations

Malgré des règles très simples à l'échelle locale, le jeu de la vie peut amener à des structures complexes et surprenantes à une échelle macroscopique. Il est même possible de créer un jeu de la vie en jeu de la vie ou de simuler une machine de Turing à condition d'utiliser une grille suffisamment grande et de choisir la bonne configuration de départ.

Ici nous nous contenterons de présenter quelques observations intéressantes obtenues suite aux essais réalisés.

4.1 Structures stables

Les structures stables sont des motifs qui ne changent plus une fois apparus. Le carré constitué de 4 cellules est l'exemple le plus simple de figure stable. Il est appelé bloc (voir figure 3).

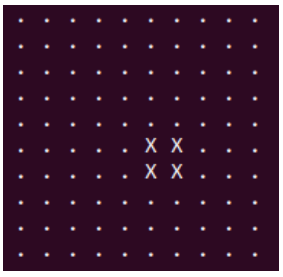

FIGURE 3 – Exemple de structure stable : le bloc

En choisissant une grille de taille 10^2 dans un univers torique avec 25% de cellules vivantes au départ, on obtient une autre figure stable qui s'appelle le bateau illustré figure 4.

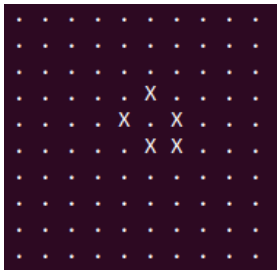

FIGURE 4 – Exemple de structure stable : le bateau

Avec une initialisation de 35% de cellules vivantes dans un univers torique, on obtient une autre figure stable appelée ruche. (voir figure 5).

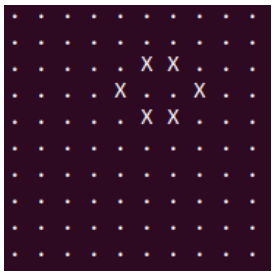

FIGURE 5 – Exemple de structure stable : la ruche

Il existe une multitude de structures stables mais celles-ci sont les plus courantes.

4.2 Structures oscillantes

Les structures oscillantes sont des motifs immobiles dont la configuration se répète selon une certaine période. Par exemple, avec 30% de cellules initiales vivantes dans un univers torique, on obtient 2 clignotants au bout d'un certain nombre d'itérations (voir figure 6). Ceux-ci oscillent avec une période de 2.

Le pulsar représenté figure 7 est un oscillateur de période 3. Il nécessite un certain nombre d'itérations pour se mettre en place.

En figure 8, un autre exemple d'oscillateur appelé galaxie est représenté. La galaxie a une période de 8.

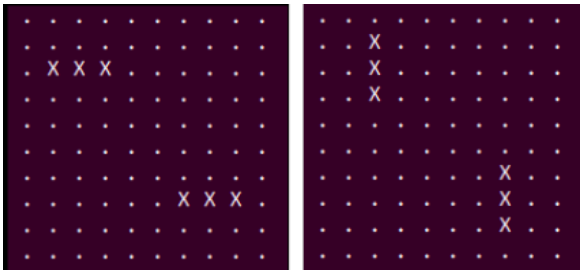

FIGURE 6 – Exemple de structure oscillante de période 2 : le clignotant

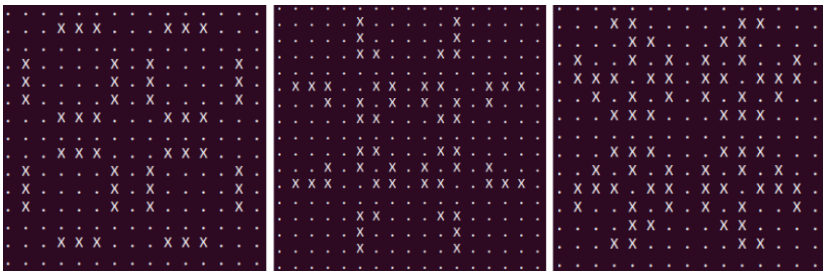

FIGURE 7 – Exemple de structure oscillante de période 3 : le pulsar

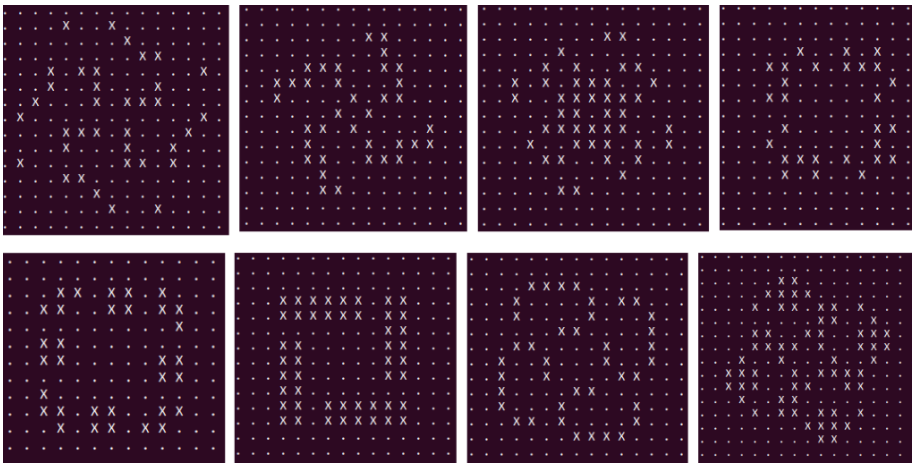

FIGURE 8 – Exemple de structure oscillante de période 8 : la galaxie

Il existe également d'autres structures oscillantes plus grandes et plus complexes. Par exemple, certaines produisent un nombre infini de gliders, elles sont appelées canons à gliders.

4.3 Gliders

Un glider est un motif qui se déplace d'une case en diagonale sur la grille selon une période de 4 (voir figure 9). C'est la plus petite structure mouvante du jeu. Elle est composée de cinq cellules comprises dans un carré de trois cellules de côté. Dans un univers torique, le glider se déplace à l'infini tandis qu'il devient un bloc dans un univers non torique.

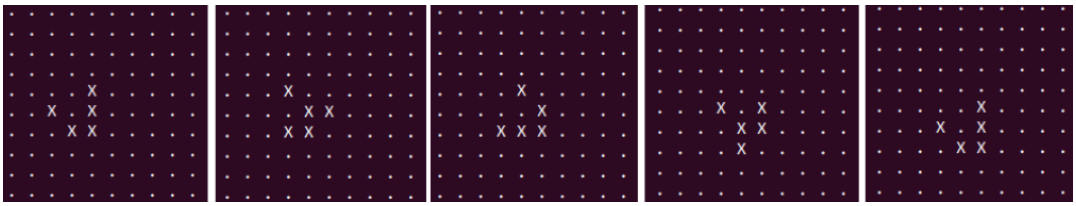

FIGURE 9 – Déplacement d'un glider en quatre étapes

D'autres figures mobiles plus grandes existent comme les vaisseaux et l'oie du Canada par exemple. On peut également citer les puffeurs qui se déplacent tout en laissant des débris derrière eux.

4.4 Étude de la décroissance cellulaire

Nous allons étudier l'évolution du nombre de cellules vivantes à partir de différentes configurations initiales aléatoires. L'étude est faite sur une grille de taille 10^2 pour 100 itérations. Le tableau 1 présente les résultats obtenus dans un univers torique tandis que le tableau 2 présente les résultats dans un univers non torique.

pourcentage initial de cellules vivantes	nombre final de cellules vivantes
0	0
5	0
10	0
15	0
20	0
25	5
30	6
35	6
40	0
45	0
50	0
55	6
60	0
65	6
70	0
75	0
80	0
85	0
90	0
95	0
100	0

TABLE 1 – Nombre de cellules vivantes après 100 itérations dans une grille de taille $10 * 10$ dans un univers torique

pourcentage initial de cellules vivantes	nombre final de cellules vivantes
0	0
5	0
10	0
15	0
20	4
25	4
30	13
35	0
40	4
45	4
50	5
55	0
60	11
65	12
70	0
75	0
80	4
85	4
90	0
95	0
100	0

TABLE 2 – Nombre de cellules vivantes après 100 itérations dans une grille de taille $10 * 10$ dans un univers non torique

Dans toutes les expériences réalisées, les cellules se retrouvent finalement éteintes ou regroupées en structures stables telles que des blocs ou des ruches. Il n'était donc pas nécessaire de faire plus de 100 itérations. On constate que dans un univers torique, il est difficile d'échapper à une extinction totale des cellules car les cellules situées aux extrémités ont plus de voisins que dans un univers non torique, elles sont par conséquent plus exposées à la surpopulation. Le nombre de cellules survivantes après 100 itérations est globalement plus élevé dans un univers non torique.

Il semble que le pourcentage idéal pour obtenir le plus de cellules à l'état final se situe entre 20 et 65%. En dessous de 20% de cellules au départ, les cellules auront tendance à s'éteindre à cause de l'isolement. Au dessus de 65% de cellules de départ, on assiste généralement à une extinction brutale

des cellules à cause de la surpopulation. Cela ressemble à ce que l'on pourrait observer en biologie lorsque l'on étudie les populations de certaines espèces vivantes.

Les résultats obtenus dans les tableaux montrent certaines tendances qui sont à nuancer car nous avons testé un nombre très limité de configurations de départ. De plus, l'état final du système est très sensible aux conditions de départ. En effet, la configuration finale peut changer complètement suite à une petite modification de la situation initiale.

Dans tous les essais réalisés, le nombre de cellules se stabilise au cours du temps. Mais ce n'est pas tout le temps le cas dans le jeu de la vie. Au cours du temps, le nombre de cellules vivantes peut se stabiliser, diminuer jusqu'à devenir nul ou encore augmenter à l'infini. C'est le cas si la grille contient un canon à gliders par exemple. Les figures ayant une croissance infinie sont appelées *spacefiller*. Par exemple, le *spacefiller* représenté figure 10 se nomme `Max` et a été découvert en 1995. Il remplit l'espace d'un motif rayé qui est localement stable. C'est le *spacefiller* dont la croissance est la plus rapide.

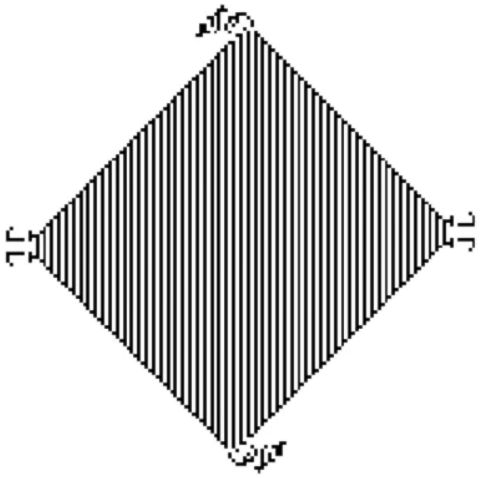

FIGURE 10 – Représentation du *spacefiller* Max

5 Conclusion

Le jeu de la vie est un système très facile à simuler qui illustre bien le concept d'émergence. En effet, pour une règle d'évolution locale extrêmement simple, on peut obtenir des comportements globaux très complexes, avec apparition et développement de structures stables, oscillantes et mêmes mobiles, plus ou moins évoluées. L'apparition de ces différentes structures est difficile à anticiper à partir de ces quelques règles fixées au départ et une infime variation de la situation initiale peut aboutir à des résultats complètement différents.

Bibliographie

[1] Voisinage de Moore [consulté le 18/03/21]
https://fr.wikipedia.org/wiki/Voisinage_de_Moore

[2] David Louapre, ScienceEtonnante, *Le Jeu de la Vie* [consulté le 18/03/21]
https://www.youtube.com/watch?v=S-W0NX97DB0

[3] *Jeu de la Vie* [consulté le 18/03/21]
http://ressources.univ-lemans.fr/AccesLibre/UM/Pedago/physique/02/recre/conway.html

[4] Nazim Fatès, *À la découverte des automates cellulaires* [consulté le 18/03/21]
https://interstices.info/a-la-decouverte-des-automates-cellulaires/

[5] *Machines arithmétiques* [consulté le 20/03/21]
http://math.pc.vh.free.fr/divers/life/machines.htm

[6] *Jeu de la Vie* [consulté le 20/03/21]
https://www-fourier.univ-grenoble-alpes.fr/sites/default/files/poster_jdv_v1.pdf

Annexes

1 Code C du jeu de la vie

```c
//TP4 : jeu de la vie

/* Bibliothèques */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/* Period parameters */
#define N 624
#define M 397
#define MATRIX_A 0x9908b0dfUL   /*constant vector a*/
#define UPPER_MASK 0x80000000UL /*most significant w-r bits*/
#define LOWER_MASK 0x7fffffffUL /*least significant r bits*/


//-----------------------------------------------------------//
// Code de Makoto Mastumoto pour le Mersenne Twister        //
//-----------------------------------------------------------//

/* the array for the state vector  */
static unsigned long mt[N];
/* mti==N+1 means mt[N] is not initialized */
static int mti=N+1;

/* initializes mt[N] with a seed */
void init_genrand(unsigned long s)
{
    mt[0]= s & 0xffffffffUL;
    for (mti=1; mti<N; mti++) {
        mt[mti] =
      (1812433253UL * (mt[mti-1] ^ (mt[mti-1] >> 30)) + mti);
       /*See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier*/
       /*In the previous versions, MSBs of the seed affect*/
       /*only MSBs of the array mt[].*/
       /*2002/01/09 modified by Makoto Matsumoto*/
       mt[mti] &= 0xffffffffUL;
       /*for >32 bit machines*/
    }
}

/* initialize by an array with array-length */
/* init_key is the array for initializing keys */
/* key_length is its length */
/* slight change for C++, 2004/2/26 */
void init_by_array(unsigned long init_key[], int key_length)
{
    int i, j, k;
```

```
48      init_genrand(19650218UL);
49      i=1; j=0;
50      k = (N>key_length ? N : key_length);
51      for (; k; k--) {
52          mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) *
    1664525UL))
53            + init_key[j] + j; /* non linear */
54          /* for WORDSIZE > 32 machines */
55          mt[i] &= 0xffffffffUL;
56          i++; j++;
57          if (i>=N) { mt[0] = mt[N-1]; i=1; }
58          if (j>=key_length) j=0;
59      }
60      for (k=N-1; k; k--) {
61          mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) *
    1566083941UL))
62            - i; /* non linear */
63          /* for WORDSIZE > 32 machines */
64          mt[i] &= 0xffffffffUL;
65          i++;
66          if (i>=N) { mt[0] = mt[N-1]; i=1; }
67      }
68      /* MSB is 1; assuring non-zero initial array */
69      mt[0] = 0x80000000UL;
70  }
71
72  /* generates a random number on [0,0xffffffff]-interval */
73  unsigned long genrand_int32(void)
74  {
75      unsigned long y;
76      static unsigned long mag01[2]={0x0UL, MATRIX_A};
77      /* mag01[x] = x * MATRIX_A  for x=0,1 */
78
79      if (mti >= N) { /* generate N words at one time */
80          int kk;
81          /* if init_genrand() has not been called, */
82          if (mti == N+1)
83              init_genrand(5489UL);
84              /* a default initial seed is used */
85
86          for (kk=0;kk<N-M;kk++) {
87              y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
88              mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
89          }
90          for (;kk<N-1;kk++) {
91              y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
92              mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0
    x1UL];
93          }
```

```
94          y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
95          mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];
96
97          mti = 0;
98      }
99
100     y = mt[mti++];
101
102     /* Tempering */
103     y ^= (y >> 11);
104     y ^= (y << 7) & 0x9d2c5680UL;
105     y ^= (y << 15) & 0xefc60000UL;
106     y ^= (y >> 18);
107
108     return y;
109 }
110
111 /* generates a random number on [0,0x7fffffff]-interval */
112 long genrand_int31(void)
113 {
114     return (long)(genrand_int32()>>1);
115 }
116
117 /* generates a random number on [0,1]-real-interval */
118 double genrand_real1(void)
119 {
120     return genrand_int32()*(1.0/4294967295.0);
121     /* divided by 2^32-1 */
122 }
123
124 /* generates a random number on [0,1)-real-interval */
125 double genrand_real2(void)
126 {
127     return genrand_int32()*(1.0/4294967296.0);
128     /* divided by 2^32 */
129 }
130
131 /* generates a random number on (0,1)-real-interval */
132 double genrand_real3(void)
133 {
134     return (((double)genrand_int32()) + 0.5)
    *(1.0/4294967296.0);
135     /* divided by 2^32 */
136 }
137
138 /*generates a random number on [0,1) with 53-bit resolution*/
139 double genrand_res53(void)
140 {
141     unsigned long a=genrand_int32()>>5, b=genrand_int32()>>6;
```

```
142     return(a*67108864.0+b)*(1.0/9007199254740992.0);
143 }
144 /* These real versions are due to Isaku Wada, 2002/01/09
        added */
145
146
147 //-------------------------------------------------------------//
148 // Code du jeu de la vie                                       //
149 //-------------------------------------------------------------//
150
151 //-------------------------------------------------------------//
152 //creationGrille : création grille taille n*n et               //
153 //                 initialisation                              //
154 //                 vide (avec des points)                      //
155 //                                                             //
156 //En entrée : n : entier égal à la taille du tableau          //
157 //                                                             //
158 //En  sortie : tableau de caractères créé                     //
159 //-------------------------------------------------------------//
160
161 char ** creationGrille (int n)
162 {
163   int     i, j, test = 1;
164   char ** grille  = NULL ;
165
166   grille = malloc ( n * sizeof (char *) );
167
168   // test réussite allocation mémoire
169   if ( grille != NULL )
170   {
171     for ( i = 0; i < n; i ++)
172     {
173       grille [i] = malloc ( n * sizeof (char) ) ;
174
175       // test réussite allocation mémoire
176       if ( grille [i] == NULL )
177       {
178         test = 0;
179       }
180     }
181   }
182
183   // en cas d'erreur d'allocation mémoire
184   if ( grille == NULL || test == 0 )
185   {
186     printf ("Erreur allocation mémoire\n");
187     //quitte le programme
188     exit (0);
189   }
```

```
190
191   // initialisation de la grille avec des points
192   for (i = 0; i < n; i ++)
193   {
194     for (j = 0; j < n; j ++)
195     {
196       grille [i][j] = '.';
197     }
198   }
199
200   return grille;
201 }
202
203 //-------------------------------------------------------------//
204 //afficheGrille : affiche la grille de taille n*n              //
205 //                                                             //
206 //En entrée : grille : tableau 2D à afficher                   //
207 //             n : entier égal à la taille du tableau          //
208 //                                                             //
209 //En  sortie : void                                           //
210 //-------------------------------------------------------------//
211
212 void afficheGrille (char ** grille, int n)
213 {
214   int i, j;
215
216   for(i = 0; i < n ; i ++)
217   {
218     for(j = 0; j < n; j ++)
219     {
220       printf ("%2c", grille [i][j]);
221     }
222     printf ("\n");
223   }
224   printf ("\n");
225 }
226
227 //-------------------------------------------------------------//
228 //initGlider : initialise la grille de taille n*n avec un      //
229 //              glider                                         //
230 //                                                             //
231 //En entrée : grille : tableau 2D contenant la grille          //
232 //             n : entier égal à la taille du tableau          //
233 //                                                             //
234 //En  sortie : grille : tableau modifié                       //
235 //-------------------------------------------------------------//
236
237 char ** initGlider (char ** grille, int n)
238 {
```

```
239   grille [1][1] = 'X';
240   grille [2][2] = 'X';
241   grille [3][0] = 'X';
242   grille [3][1] = 'X';
243   grille [3][2] = 'X';
244 
245   return grille;
246 }
247 
248 //-------------------------------------------------------------//
249 //initGalaxie : initialise la grille de taille 50^2 avec   //
250 //              une galaxie                                  //
251 //                                                           //
252 //En entrée : grille : tableau 2D contenant la grille      //
253 //            n : entier égal à la taille du tableau       //
254 //                                                           //
255 //En  sortie : grille : tableau modifié                    //
256 //-------------------------------------------------------------//
257 
258 char ** initGalaxie (char ** grille, int n)
259 {
260   if (n != 50)
261   {
262     printf ("Grille trop petite pour contenir une galaxie !\n
    ");
263     // quitte le programme
264     exit (0);
265   }
266 
267   int i, j;
268 
269   // initialisation de la galaxie
270   for (i = 20; i < 22; i ++)
271   {
272     for (j = 20; j < 26; j++)
273     {
274       grille [i][j] = 'X';
275     }
276   }
277 
278   for (i = 27; i < 29; i ++)
279   {
280     for (j = 23; j < 29; j++)
281     {
282       grille [i][j] = 'X';
283     }
284   }
285 
286   for (i = 23; i < 29; i ++)
```

```
287   {
288     for (j = 20; j < 22; j++)
289     {
290       grille [i][j] = 'X';
291     }
292   }
293 
294   for (i = 20; i < 26; i ++)
295   {
296     for (j = 27; j < 29; j++)
297     {
298       grille [i][j] = 'X';
299     }
300   }
301 
302   return grille;
303 }
304 
305 
306 //-------------------------------------------------------------//
307 //initPulsar : initialise la grille de taille 50^2 avec     //
308 //             un pulsar                                     //
309 //                                                           //
310 //En entrée : grille : tableau 2D contenant la grille       //
311 //            n : entier égal à la taille du tableau         //
312 //                                                           //
313 //En  sortie : grille : tableau modifié                     //
314 //-------------------------------------------------------------//
315 
316 char ** initPulsar (char ** grille, int n)
317 {
318   if (n != 50)
319   {
320     printf ("Grille trop petite pour contenir un pulsar !\n")
     ;
321     //quitte le programme
322     exit (0);
323   }
324 
325   int i , j = 20;
326 
327   // initialisation du pulsar
328   for (i = 20; i < 25; i ++)
329   {
330     grille [i][j] = 'X';
331   }
332 
333   j = 24;
334 
```

```
335   for (i = 20; i < 25; i ++)
336   {
337     grille [i][j] = 'X';
338   }
339 
340   grille [20][22] = 'X';
341   grille [24][22] = 'X';
342 
343   return grille;
344 }
345 
346 //-------------------------------------------------------------//
347 //initRandom : initialise la grille de taille n*n              //
348 //              aléatoirement avec une proportion choisie      //
349 //               de cellules vivantes                          //
350 //                                                             //
351 //En entrée : grille : tableau 2D contenant la grill           //
352 //             n : entier égal à la taille du tableau          //
353 //             prop : réel égal à la prop choisie              //
354 //             de cellules                                     //
355 //             vivantes au début du jeu                        //
356 //                                                             //
357 //En  sortie : grille : tableau modifié                        //
358 //-------------------------------------------------------------//
359 
360 char ** initRandom (char ** grille, int n, float prop)
361 {
362   double rdm;
363   int    i, j;
364 
365   //initialisation aléatoire
366   for (i = 0; i < n ; i ++)
367   {
368     for (j = 0; j < n; j ++)
369     {
370       rdm = genrand_real2();
371 
372       if (rdm < prop)
373       {
374         grille [i][j] = 'X';
375       }
376     }
377   }
378 
379   return grille;
380 }
381 
382 
383 //-------------------------------------------------------------//
```

```
384 //libereGrille : libère la mémoire de la grille de          //
385 //                taille n*n                                  //
386 //                                                           //
387 //En entrée : grille : tableau 2D contenant la grille       //
388 //             n : entier égal à la taille du tableau        //
389 //                                                           //
390 //En  sortie : void                                         //
391 //-----------------------------------------------------------//
392
393 void libereGrille (char ** grille, int n)
394 {
395   int i;
396
397   if (grille != NULL)
398   {
399     for (i = 0; i < n; i ++)
400     {
401       free (grille[i]);
402     }
403     free (grille);
404   }
405 }
406
407 //-----------------------------------------------------------//
408 //indiceValide : renvoie 1 si l'indice [k][l] de la cellule//
409 //               voisine de la case considérée [i][j]        //
410 //               est valide, 0 sinon                         //
411 //               L'indice est valide si la case voisine      //
412 //               de [i][j] n'est pas [i][j] et si elle       //
413 //               ne se trouve pas en dehors de la grille     //
414 //               Cette fonction sert uniquement pour         //
415 //               l'univers non torique                       //
416 //                                                           //
417 //                                                           //
418 //En entrée : grille : tableau 2D contenant la grille       //
419 //             n : entier égal à la taille du tableau        //
420 //             i : ligne de la case considérée               //
421 //             j : colonne de la case considérée             //
422 //             k : ligne de la case voisine                  //
423 //             l : colonne de la case voisine                //
424 //                                                           //
425 //En  sortie : entier 1 ou 0                                //
426 //-----------------------------------------------------------//
427
428 int indiceValide (char ** grille, int n, int i, int j, int k,
      int l)
429 {
430   // cellule étudiée
431   if ( k == i && l == j )
```

```
  {
    return 0;
  }
  // cellule en dehors de la grille
  else if ( k < 0 || l < 0 || k >= n || l >= n )
  {
    return 0;
  }
  else
  {
    return 1;
  }
}


//-------------------------------------------------------------//
//nbVoisins : renvoie le nombre de voisins vivants de          //
//            la case d'indice [i][j] de la grille             //
//            dans un univers non torique                      //
//                                                             //
//                                                             //
//En entrée : grille : tableau 2D contenant la grille          //
//            n : entier égal à la taille du tableau           //
//            i : ligne de la case considérée                  //
//            j : colonne de la case considérée                //
//                                                             //
//En  sortie : nombre de voisins (entier)                      //
//-------------------------------------------------------------//

int nbVoisins (char ** grille, int n, int i, int j)
{
  int nv = 0, k, l, test = 0 ;

  for (k = (i - 1); k <= (i + 1); k ++)
  {
    for (l = (j - 1); l <= (j + 1); l ++)
    {
      test = indiceValide (grille, n, i, j, k , l);

      // si l'indice du voisin est valide et si le voisin est
    vivant
      if ( test  == 1 && grille [k][l] == 'X' )
      {
        nv ++;
      }
    }
  }

  return nv;
```

```
480 }
481
482
483 //-----------------------------------------------------------//
484 //nbVoisinsTore  : renvoie le nombre de voisins vivants     //
485 //                 de la case d'indice [i][j] de la         //
486 //                 grille dans un univers torique           //
487 //                                                          //
488 //                                                          //
489 //En entrée : grille : tableau 2D contenant la grille       //
490 //            n : entier égal à la taille du tableau        //
491 //            i : ligne de la case considérée               //
492 //            j : colonne de la case considérée             //
493 //                                                          //
494 //En  sortie : nombre de voisins (entier)                   //
495 //-----------------------------------------------------------//
496
497 int nbVoisinsTore (char ** grille, int n, int i, int j)
498 {
499   int nv = 0, k, l, kv, lv;
500
501   for (k = (i - 1); k <= (i + 1); k ++)
502   {
503     for (l = (j - 1); l <= (j + 1); l ++)
504     {
505       kv = k;
506       lv = l;
507
508       // adaptation de l'indice dans l'univers torique
509       if ( k < 0 )
510       {
511         kv = k + n;
512       }
513
514       if ( l < 0 )
515       {
516         lv = l + n;
517       }
518
519       if ( k >= n )
520       {
521         kv = k % n;
522       }
523
524       if ( l >= n )
525       {
526         lv = l % n;
527       }
528
```

```
529       // si l'indice du voisin n'est pas la cellule actuelle
      et si le voisin est vivant
530       if ( ( kv != i || lv != j ) && grille [kv][lv] == 'X' )
531       {
532         nv ++;
533       }
534     }
535   }
536
537   return nv;
538 }
539
540
541 //-------------------------------------------------------------//
542 //jeu : modifie la grille2 à partir de grille1 en suivant  //
543 //      les règles du jeu de la vie                          //
544 //                                                           //
545 //                                                           //
546 //En entrée : grille1 : tableau 2D contenant la grille      //
547 //            à l'instant t grille2 :tableau 2D              //
548 //            contenant la grille à l'instant t+1            //
549 //            taille : entier égal à la taille du tableau    //
550 //            tore : entier valant 1 si on est dans un       //
551 //            univers torique et 0 dans le cas contraire     //
552 //                                                           //
553 //En  sortie : void (passage par référence)                  //
554 //-------------------------------------------------------------//
555
556 void jeu (char ** grille1, char ** grille2, int taille, int
      tore)
557 {
558   int i, j, nv = 0;
559
560   for (i = 0; i < taille ; i ++)
561   {
562     for (j = 0; j < taille; j ++)
563     {
564       // univers non torique
565       if (tore == 0)
566       {
567         nv = nbVoisins (grille1, taille, i, j);
568       }
569       // univers torique
570       else
571       {
572         nv = nbVoisinsTore (grille1, taille, i, j);
573       }
574
575       // la cellule vivante survit si elle a 2 ou 3 voisins
```

```
576       if ( grille1 [i][j] == 'X' && (nv == 2 || nv == 3) )
577       {
578         grille2 [i][j] = 'X';
579       }
580       // la cellule morte nait si elle a 3 voisins
581       else if ( grille1 [i][j] == '.' &&  nv == 3 )
582       {
583         grille2 [i][j] = 'X';
584       }
585       // la cellule meurt ou reste morte dans les autre cas
586       else
587       {
588         grille2 [i][j] = '.';
589       }
590     }
591   }
592 }
593 
594 //-------------------------------------------------------------//
595 // programme principal                                         //
596 //-------------------------------------------------------------//
597 
598 int main (int argc, char *argv[])
599 {
600   // initialisation du Mersenne Twister
601   unsigned long init[4] = {0x123, 0x234, 0x345, 0x456},
      length = 4;
602   init_by_array(init, length);
603 
604   // déclaration des variables
605   int     tailleGrille, choix, i, tore, debug = 1, iter;
606   float   prop;
607   char ** grille1;
608   char ** grille2;
609 
610 
611   printf ("Bienvenue dans le jeu de la vie !\n\n");
612 
613   printf ("Veuillez choisir une option pour le type d'univers
      :\n\n");
614   printf ("1 : univers torique \n");
615   printf ("2 : univers non torique \n");
616   printf ("\nVotre choix : ");
617 
618   scanf ("%d", &choix);
619 
620   // choix du type d'univers
621   switch (choix)
622   {
```

```
623     case 1 :
624       tore = 1;
625     break;
626 
627     case 2 :
628       tore = 0;
629     break;
630 
631     default :
632       printf ("Erreur de saisie\n");
633       //quitte le programme
634       exit (0);
635     break;
636   }
637 
638   printf ("\n");
639 
640   printf ("Veuillez choisir la taille de la grille de jeu :\n
      \n");
641   printf ("1 : grille de taille 10*10 \n");
642   printf ("2 : grille de taille 50*50 \n");
643   printf ("\nVotre choix : ");
644 
645   scanf ("%d", &choix);
646 
647   // choix de la taille de la grille
648   switch (choix)
649   {
650     case 1 :
651       tailleGrille = 10;
652     break;
653 
654     case 2 :
655       tailleGrille = 50;
656     break;
657 
658     default :
659       printf ("Erreur de saisie\n");
660       //quitte le programme
661       exit (0);
662     break;
663   }
664 
665   printf ("\n");
666 
667   // création de 2 grilles de taille tailleGrille
668   grille1 = creationGrille (tailleGrille);
669   grille2 = creationGrille (tailleGrille);
670 
```

```
671
672
673   printf ("Veuillez choisir l'initialisation de la grille de
       jeu :\n\n");
674   printf ("1 : glider \n");
675   printf ("2 : galaxie (seulement pour une grille 50*50)\n");
676   printf ("3 : pulsar (seulement pour une grille 50*50)\n");
677   printf ("4 : aléatoire \n");
678   printf ("\nVotre choix : ");
679
680   scanf ("%d", &choix);
681
682   printf ("\n");
683
684   // choix de l'initialisation
685   switch (choix)
686   {
687     case 1 :
688       initGlider (grille1, tailleGrille);
689     break;
690
691     case 2 :
692       initGalaxie (grille1, tailleGrille);
693     break;
694
695     case 3 :
696       initPulsar (grille1, tailleGrille);
697     break;
698
699     case 4 :
700       printf ("Veuillez entrer la proportion de cellules
      vivantes au départ : ");
701
702       scanf ("%f", &prop);
703
704       if ( prop < 0 || prop > 1 )
705       {
706         printf ("Le nombre saisi doit etre compris entre 0 et
       1 \n");
707         exit (0);
708       }
709       else
710       {
711         printf ("\n");
712         initRandom (grille1, tailleGrille, prop);
713       }
714     break;
715
716     default :
```

```
717       printf ("Erreur de saisie\n");
718       //quitte le programme
719       exit (0);
720     break;
721   }
722
723   // choix du nombre d'itérations
724   printf ("Veuillez choisir le nombre d'itérations : ");
725
726   scanf ("%d", &iter);
727
728
729   printf ("\nDébut du jeu :\n");
730
731   afficheGrille (grille1, tailleGrille);
732
733   // pause de 1 seconde pour voir l'état initial
734   usleep (1000000);
735
736   for (i = 0; i < iter; i ++)
737   {
738     if ( i % 2 == 0)
739     {
740       jeu (grille1, grille2, tailleGrille, tore);
741       afficheGrille (grille2, tailleGrille);
742     }
743     else
744     {
745       jeu (grille2, grille1, tailleGrille, tore);
746       afficheGrille (grille1, tailleGrille);
747     }
748
749     // pause de 0.3 secondes en mode debug
750     if (debug == 1)
751     {
752       usleep (300000);
753     }
754   }
755
756   printf ("Fin du jeu après %d itérations\n", iter);
757
758
759   // libération mémoire
760   libereGrille (grille1, tailleGrille);
761   libereGrille (grille2, tailleGrille);
762
763   return 0;
764 }
```