Brinda MUZAKARE

Jean de Dieu NDUWAMUNGU IMA 3 Groupe TD 2

12/01/2018

POLYTECH LILLE

PROJET : JEU DE LA VIE

Introduction

Le jeu de la vie est un processus simulant sur une grille le cycle de vie ou de mort de cellules à partir d'un motif initial. Une cellule n'a que deux états possibles : 1 pour vivante et 0 pour morte.

Le jeu de la vie repose sur le principe d'évolution de la grille dans le temps. À chaque étape, appelée génération, les cellules évoluent en fonction de leur voisinage (chaque cellule a 8 cellules voisines).

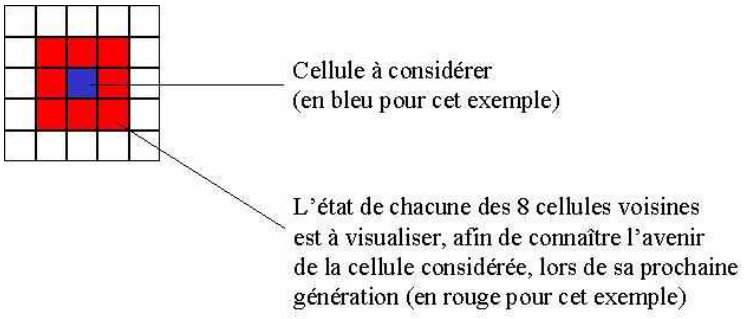

Figure 1 Source : http://cypris.fr/loisirs/le_jeu_de_la_vie/jeu_de_la_vie.htm

Si une cellule vivante possède 0 ou 1 voisins alors elle meurt, de même qu'une cellule entourée de 4 voisins ou plus. Si elle est entourée de 2 ou 3 voisins alors elle reste en vie à la prochaine génération et une cellule naitra dans une case vide entourée de 3 cellules vivantes. (voir figure 2)

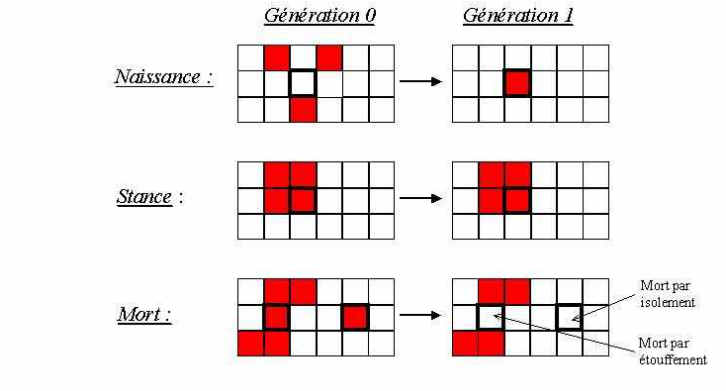

Figure 2 Source : http://cypris.fr/loisirs/le_jeu_de_la_vie/jeu_de_la_vie.htm

Notre objectif est de programmer ce jeu dans le langage de programmation C en faisant appel à des notions vus en cours d'informatique notamment les principes du codage en C et les redirections.

Dans ce rapport nous tacherons de distinguer les différentes avancement de notre programme et de justifier le choix de nos méthodes.

I. CONCEPTION DU PROGRAMME

Afin de simuler le « Jeu de la vie » nous avons choisi pour notre programme d'attribuer la valeur 1 à une cellule vivante et la valeur 0 pour représenter une case vide.

Initialisation de la grille

On a eu besoin de deux tableaux. Le premier tableau est celui de notre matrice, de notre grille initiale. Celle-ci peut être initialisée avec des valeurs aléatoires en utilisant une fonction '***hasard'*** et '***remplir'*** programmées pour fournir une valeur aléatoire entre 0 et 1, ou dans un autre cas on peut charger une grille préenregistrée. Une fois la grille chargée on a remarqué qu'on a besoin d'une deuxième table en plus pour enregistrer l'évolution suivante de notre table initiale. La fonction qui va effectuer les opérations de compte de cases voisines autour de la cellule que l'on va tester est ***'test_tab'***. Elle prend en paramètre la taille de la grille et deux tableaux. Le premier tableau correspond à notre grille au départ et le second une table de même taille qui va nous servir par après.

Affichage des évolutions

Deux possibilités d'affichage sont offerts à l'utilisateur : le premier où l'utilisateur passe explicitement d'un affichage au suivant. Nous avons mis une simple boucle ***while*** qui va s'exécuter tant que l'utilisateur n'aura pas décidé d'arrêter le programme. Pour les évolutions automatiques nous avons défini dès le départ le nombre maximum d'évolutions. La première ligne de notre grille correspond au nombre de génération.

Comment compter le nombre de cases autour d'une cellule?

Une case a 8 cellule autour d'elle. Nous avons utilisé deux boucles ***'for'*** pour parcourir le tableau ***'tab'*** de taille ***'i j'***. Arrivé à l'élément ***'tab[i][j]'*** nous avons besoin de compter le nombre des voisins à cette case sans toutefois la modifier ou perdre sa valeur. Pour cela nous avons pensé à utiliser deux autres variables ***'k'*** et ***'l'*** qui vont parcourir les 8 cases autour de notre élément. K va parcourir le tableau tab de i-1 à i+1 et l de j-1 à j+1.

Comme on nous a demandé d'utiliser une variante périodique il faut que notre programme, lorsque l'on sera sur le ligne 0, prenne les éléments de la dernière ligne et lorsque l'on sera sur la colonne 0 prenne les éléments de la dernière colonne et vice versa. Pour cela on a établi des fonctions ***'if'*** suivant les différents cas. Et à chaque fois que l'on va rencontrer une cellule vivante donc 1 dans notre programme une variable '***cmp'*** sera incrémentée. Cela veut dire que notre variable **cmp** peut être incrémentée 9 fois au maximum si elle rencontre que des 1 en parcourant de ***i-1*** à ***i+1*** et k-1 à k+1. D'où lorsque l'on va sortir de la boucle il faut réduire de 1 **cmp** si la valeur de la case vaut 1.

Si notre élément test (valeur de la case) vaut 1, ce qui correspond à une cellule vivante, on applique les conditions de survie (si elle a 2 ou 3 voisins) ou de mort(si elle 1 ou moins, 4 ou plus voisines vivantes).

Si notre élément test vaut 0, ce qui correspond à une case vide, on applique les conditions de naissance (si elle a 3 voisines) ou cette dernière demeure à 0 (si elle n'a pas exactement 3 cellules voisines).

Après cette étape on peut afficher notre tableau 'tab1'.

Ce dernier correspond à la nouvelle génération. On a ajouté une fonction '**copietab'** qui va copier le tableau tab1 et remplacer la génération initiale dont on a plus besoin. Le processus va se répéter autant de fois que la limite que nous avons établi dans notre fonction main ne sera pas atteinte.

Exemples de fonctionnement du programme test_tab

Exemple 1

On a un tableau déjà établi et notre fonction va le parcourir pour tester les différents éléments un à un. Supposons que l'élément que notre programme va tester est la toute première case de la grille de la grille l'élément tab[0][0].

Nos deux boucles for vont commencer à tester l'élément ***tmp***=tab[0][0] et ***cmp*** qui va s'incrémenter lorsque on rencontre une cellule vivante vaut 0. Donc ici i=0 et j=0

On rentre dans deux boucles for (k et l) qui vont parcourir autour de ***tmp*** de i-1 à i+1 et de j-1 à j+1. Comme k=i-1 et que i vaut O=> k vaut -1 au départ on prend la dernière ligne => k=M-1(dernière ligne). De plus l vaut -1 => On prend la dernière colonne => l=N-1. On va donc tester en premier l'élément tab[M-1][N-1]

On remarque que les valeurs de k et 'l' ont été modifiées d'où il faut les sauvegarder dans une variable pour pouvoir les récupérer et les réutiliser à la prochaine itération.

Si tab[M-1][N-1] vaut 1 on incrémente ***cmp***.

- 2è itération des deux boucles for k et l. Maintenant k vaut -1 et l vaut 0 => comme k<0 on prend k=M-1. Donc l'élément à tester est tab[M-1][0] et on incrémente cmp suivant que l'on rencontre 1 ou 0.
- 3è itération k vaut -1 et l vaut 1 => on teste tab[M-1][1]
- 4è itération k vaut 0 et l vaut -1 => on teste tab[0][N-1]
- 5è itération k vaut 0 et l vaut 0 => on teste tab[0][0]
- 6è itération k vaut 0 et l vaut 1 => on teste tab[0][1]
- 7è itération k vaut 1 et l vaut -1 => on teste tab[1][N-1]
- 8è itération k vaut 1 et l vaut 0 => on teste tab[1][0]
- 9è itération k vaut 1 et l vaut 1 => on teste tab[1][1]
- On a fait 9 tests alors que les cases voisines à notre cellule tmp sont au nombre de 8. D'où il faut ***cmp=cmp-1*** si la cellule tab[0][0] avait pour valeur 1.
- On applique les conditions de naissance ou de mort
- On ajoute la valeur de tmp dans l'autre table, celle de la prochaine génération. Ici tab1

Exemple 2

Supposons que nous sommes à l'élément tab[4][5] c'est à dire tmp=tab[4][5]

1. i=4 et j=5. => Au début k=i-1=3. Et l=j-1=4
   ⇨ on teste l'élément tab[3][4]
2. on teste ensuite tab[3][5]
3. Et puis tab[3][6]
4. k=i=4 et l=j-1=4 => On teste tab[4][4]
5. Test de tab[4][5]
6. Test de tab[4][6]
7. K=i+1 => test de tab[5][4]
8. Test de tab[5][5]
9. Test de tab[5][6]

9 itérations au total.

Exemple 3

Supposons que nous sommes à l'élément tab[0][5] c'est à dire tmp=tab[4][5]

1. i=0 et j=5. => Au début k=i-1=-1. Et l=j-1=4
   ⇨ on teste l'élément tab[M-1][4]
2. on teste ensuite tab[M-1][5]
3. Et puis tab[M-1][6]
4. k=i=0 et l=j-1=4 => On teste tab[0][4]
5. Test de tab[0][5]
6. Test de tab[0][6]
7. K=i+1 => test de tab[1][4]
8. Test de tab[1][5]
9. Test de tab[1][6]

9 itérations au total.

Dans notre fichier nous avons mis trois différentes grilles pour différents test de notre programme.

II. Limitations de la solution & améliorations possibles

Les redirections ne fonctionnant pas avec le cas où l'utilisateur passe explicitement d'un affichage au suivant nous avons dû choisir entre les 2 affichages en privilégiant les redirections.

On aurait pu améliorer notre programme en utilisant une fonction qui pourrait comparer trois évolutions successives de notre tableau et ainsi déterminer le cas où on il n'y a plus de changements et donc les cas où on tourne en rond. On a écrit une fonction ***'compar_tab'*** pour comparer deux évolutions successives mais on s'est vite rendu compte qu'en ajoutant la comparaison il y'avait un problème d'affichage des états stables dans notre grille.

On aurait pu améliorer également notre programme notamment l'affichage en insérant des caractères spéciaux à la place de 1 et 0 pour un rendu meilleur.

Nous avons remarqués en écrivant notre fonction effectuant les tests 'test_tab' que lorsque nous comptons les cases autour d'une cellule il nous fallait utiliser à chaque fois des ***if*** pour que le programme puisse à fonctionner correctement.

```
if (k<0){
    k=M-1;//la ligne au dessus de la ligne 0 correspond à M-1 : variante périodique
}

if (l<0)
{
    l=N-1;//la colonne à gauche de la colonne 0 correspond à N-1 : variante périodique
}

if (k>M-1)
{
  k=0;
}

if (l>N-1)
{
  l=0;
}

if (tab[k][l]==1) {
    cmp=cmp+1;
}
```

Certes notre programme ne sera pas efficace en effectuant des comparaisons qui ne sont pas nécessaires mais nous n'avons pas pu le faire autrement en se rendant compte qu'avec les 'else if' une erreur d'identification des bonnes cases persistait.

La principale difficulté rencontré au cours de notre projet a été de rendre la grille périodique.

Les cases du milieu fonctionnant toujours sur le même principes de compté le nombre de cellules existant autour d'une case donnée, nous avons essayé de garder le même code qu'une itération simple et de trouver un autre code spécialement pour les cases situés à l'extrémité.

Nous avons buté sur une autre difficulté qui consistait à voir que les 4 cases situés dans les 4 coins de la grille avaient des comportements différents de autres cases situés à l'extrémité.

III. CONCLUSION

Finalement, nous pouvons dire que le projet de Jeu de la vie nous a fourni une expérience enrichissante. Le fait de l'avoir fait en binôme nous a permis d'échanger sur les solutions possibles d'écouter les propositions de chacun et donc de développer notre communication. Nous avons fait appel à des notions vus en cours notamment les mécanismes de shell, les redirections, les principes de codage en langage C. On a appris à respecter un cahier de charge à organiser bien nos tâches pour arriver au but de notre projet. Les étapes du cahier de charge nous ont permises d'aborder le projet d'abord de la façon la plus simple (variante non périodique) ensuite en intégrant la variante périodique. Une stratégie que nous pourrons utiliser dans nos futurs projets.