TP7. Algorithmes : automates cellulaires. Jeux de la vie.

Dimitri Galayko

Dans ce TP on écrira un programme qui modélisera le célèbre jeu de la vie. En voici le résumé des règles.

Il y a une colonie de bactérie, qui vit dans un champ quadrillé orthogonal: les cases sont rangées en YMAX lignes et en XMAX colonnes. Par convention, le coorsonnées des cases vont de 1 à XMAX sur l'axe horisontal, et de 1 à YMAX sur l'axe vertical. Dans chaque case peut vivre une bactérie; sinon, la case est vide.

La colonie évolue d'une génération à l'autre de la manière suivante.

Si à la génération n une case vide a exactement 2 voisins, à la génération $n+1$ une bactérie naît dans cette case.

Si à la génération n une bactérie a plus de 3 voisins, elle meure de la promiscuité, c.a.d., à la génération $n+1$ cette case sera vide.

Si à la génération n une bactérie a moins de 2 voisins, elle meure de la solitude, et dans la prochaine génération cette case est vide.

Dans tous les autres cas, l'état d'une case reste le même.

Dans ce TD, vous devrez écrire un programme qui définira la configuration initiale de la colonie, qui calculera son évolution et qui l'affichera à l'écran.

Pour visualiser l'évolution de la colonie, on utilisera les fonctions graphiques que nous avons développées pour l'affichage des cases en champ quadrillé. Nous possèdons donc d'un champ graphique carthésien, partitionné en XMAX par YMAX cases, chaque case occupe LX par LY pixel. Une bactérie est représentée par un motif (un dessin) dans une case du champ.

Testez l'exécutable du programme disponible dans le répertoire ≈galayko/INFO2/TP7/life

Pour écrire le programme, on créera d'abord une bibliothèque de fonctions nécessaires pour réaliser l'algorithme.

L'information sur la colonie $YMAX \times XMAX$ sera stoquée dans un tableau à deux dimensions appelé *population*, qui contiendra $YMAX \cdot XMAX$ éléments. Ce tableau sera global. Un autre tableau de taille identique appelé *buffer* représentera un tampon temporaire dans lequel sera enregistré l'état de la colognie à la prochaine génération.

Les dimensions $YMAX$ et $XMAX$ du champ quadrillé sont données via les directives #define.

1 Fonctions d'affichage

Exo 1.1. Écrivez la fonction

```
void print_cell(int x, int y){
...
}
```

qui affiche une bactérie dans une case à coordonnées logiques (x,y). La bactérie est dessinée selon un motif donnée dans le tableau global *cell*. Ce tableau a la même dimension qu'une case du champ quadrillé a de pixels (LX sur l'horisontale, LY sur la verticale). Chaque élément de ce

tableau représente un pixel, l'ensemble des éléments du tableau définissent le motif (cf. le fichier fourni).

Exo 1.2. Écrivez la fonction

```
void remove_cell(int x, int y){
...
}
```

qui efface une case à coordonnées logiques (x,y), à l'aide de la fonction *effacer_zone* de la bibliothèque graphique.

Exo 1.3. Écrivez la fonction

```
void print_population_init(){
...
}
```

qui affiche la population initiale. Cette fonction ne sera appellée qu'une seule fois, au début de l'exécution du programme.

Exo 1.4. Écrivez la fonction

```
void print_population_delta(){
...
}
```

qui affiche uniquement des changements entre l'actuelle population (donnée par le tableau *population*) et la population à la prochaine génération (donnée par le tableau *buffer*). Cette fonction sera appellée chaque fois qu'une novelle génération sera "calculée" et enregistrée dans le tableau *buffer*. Après l'affichage de la population de la nouvelle génération à l'écran, le tableau *tampon* sera copiée vers le tableau *population*.

2 Fonctions d'inspection/surveillance de la population courante

Ces fonctions permettent de connaître des informations sur la population courante. On s'en servira plus tard pour générer une nouvelle générations. Pour connaître les différentes informations, ces fonctions interrogent le tableau *population* qui contient l'information sur la population courante.

Exo 2.1. Écrivez la fonction

```
int anybody_home(int x, int y){
}
```

qui prend en argument les coordonnées d'une case, et qui retourne 1 si la case est occupée, 0 sinon.

Attention ! Les cases sont numérotées à partir de 1.

Exo 2.2. Écrivez la fonction

```
int neighbor_number(int x, int y){
}
```

qui compte le nombre de voisins d'une case spécifiée par ces coordonnées. On considère qu'une case a 4 cases voisins ; on peut également imaginer une version où on compte également les voisins dans les directions diagonales, on aura alors 9 voisins.

Faites attentions aux cases qui se trouvent aux bords du champ.

Exo 2.3. Écrivez la fonction

```
int is_dead(int x, int y){
}
```

qui retourne 1 si la bactérie se trouvant dans la case (x,y) doit mourir à la prochaine génération, 0 sinon.

Exo 2.4. Écrivez la fonction

```
int is_born(int x, int y){
}
```

si dans la case (x,y) doit naître une bactérie à la prochaine génération

3 Nouvelle génération

Comme on a déjà dit dans l'introduction, on calcule l'état de la génération future dans un autre tableau appelé *buffer* qui a les mêmes dimensions que le tableau *generation*. Ce calcul se fait en analysant l'état de la génération courante. Lorsque la configuration de la nouvelle génération sera établie, on copiera le contenu du tableau *buffer* vers le tableau *génération*.

Exo 3.1. Écrivez la fonction

```
void born(int x, int y){
}
```

qui "prend acte" de la naissance d'une bactérie dans la case (x,y), autrement dit, qui écrit 1 dans la case (x,y) du tableau *buffer*.

Exo 3.2. Écrivez la fonction

```
void dead(int x, int y){
}
```

qui, de la même manière, "prend acte" de la mort d'une bactérie se trouvant dans la case (x,y), en écrivant 0 dans la case correspondante du tableau *buffer*.

Exo 3.3. Écrivez la fonction

```
void nothing\_change(x,y){
}
```

qui "prend acte" du fait que l'état de la case (x,y) n'a pas changé entre deux génération. C.a.d., pour cette case, la valeur de "génération" est recopiée dans *buffer*.

4 Fonction renouvellement de génération

Écrivez la fonction qui calcule la configuration de la nouvelle génération et qui met à jours le tableau "generation", en utilisant les fonctions écrites précédemment.

```
void renew_generation(void){
}
```

Pour cela, vous parcourrez toutes les cases du champ de la génération courante. Si une case n'est pas vide, vous vérifierez si la bactérie doit mourir, auquel cas vous appellerez la fonction *dead(x,y)*. Si une case est vide, vous vérifierez si une bactérie doit y naître, auquel cas vous appellerez la fonction *born(x,y)*. Dans tous les autres cas, vous appellerez la fonction *nothing_changes*.

Lorsque vous aurez fait ces opérations pour toutes les cases, vous aurez créé une nouvelle population dans le tableau *buffer*. Après cela, il n'y aura qu'à copier le tableau *buffer* vers *population*.

5 Test : fonction main

Avec toutes ces fonctions écrites, la fonction main est minimaliste. Là voici:

```
int main(){

     creer_fenetre(XMAX*LX,YMAX*LX,"white","blue");

     print_population_init();
     while(1){
          renew_generation();
          attendre_clavier();
          print_population_delta();
     }
     return 0;
}
```

Cette fonction main est en fait une boucle qui imprime la "carte" de la colonie, et qui attend à ce que l'utilisateur appuie sur n'importe quelle touche pour afficher une nouvelle génération. Cela continue jusqu'à ce que l'utilisateur ne ferme la fenêtre graphique.