TP N°9-bis
Jeu de la vie de Conway

Préambule

Nous allons modéliser une simulation connue sous le nom de « jeu de la vie de Conway ». Ce jeu, inventé par le mathématicien John Conway est un automate cellulaire. Ce n'est pas vraiment un « jeu » mais plus une simulation mathématique.

Je vous invite à vous renseigner sur ce jeu sur Wikipedia :

https://fr.wikipedia.org/wiki/Jeu_de_la_vie

Ou encore à visualiser cette vidéo :

https://www.youtube.com/watch?v=S-W0NX97DB0

Il va donc s'agir de créer un tableau à deux dimensions h x l (hauteur, largeur) dans lequel chaque case sera une cellule qui aura deux valeurs possibles morte : 0 ou vivante : 1

Le tableau va évoluer dans le temps et pour suivre cette évolution à chaque étape, le plus simple est de créer deux tableaux : un tableau qui représente l'état du jeu à l'instant n-1 (le passé) et un tableau qui représente l'état du jeu à l'instant n (le présent).

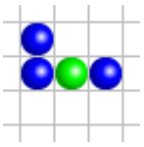
Le plus important dans ce jeu est de connaitre le « cycle de vie » de chaque cellule.

- une cellule morte possédant exactement trois voisines vivantes devient vivante (elle naît) ;
- une cellule vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt.

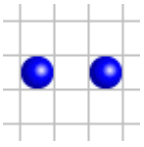
Une cellule possède exactement huit voisins dans la grille.

Exemple :

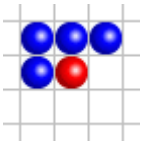
Une cellule présente à l'étape n-1 est bleue, une cellule qui naît en vert, une cellule qui meurt en rouge. (Naitre = mort à n-1, vivant à n ; Mourir = vivante à n-1, morte à l'étape n)



les trois cellules bleues étaient vivantes à l'étape n-1 donc la cellule verte naît



la cellule morte du milieu n'a que deux voisines donc elle reste morte



La cellule du milieu a quatre voisins donc elle meurt

Concernant l'affichage nous allons utiliser un mécanisme très simple :

Si une cellule est morte on affiche un espace, si elle est vivante, on affiche le caractère O (ou un autre caractère qui remplit bien l'espace)

Pour que ce jeu soit vraiment paramétrable nous offrirons la possibilité à l'utilisateur de charger un fichier de départ, dans lequel il indiquera :

// La taille de la grille :

8 // la largeur

4 // la hauteur

// La grille de départ ou chaque cellule est codée par 0 ou 1

00001000

01010101

10100000

10000000

Il existe sous linux un format simple d'image, le format pbm qui contient exactement les informations dont nous avons besoin. Nous verrons en milieu de TP comment l'utiliser.

1. Structure image

Nous allons commencer par créer une structure image_pbm dans un module image_pbm(.c/.h). Ces fichiers contiendront toutes les opérations de manipulation, d'affichage relatif à la structure pbm.

Le fichier main.c sera appelé à évoluer au fur et à mesure du projet, pour faire des tests. Au final, la fonction **`main()`** servira à lancer et afficher le jeu de la vie.

Créer une structure image_PBM. Elle contient

- width : la largeur de l'image
- height : la hauteur de l'image
- pixels : un pointeur vers un unsigned char qui va contenir un tableau de 0 et de 1

Note : pour ce TP on utilisera un tableau « aplati », plus simple à gérer. Le tableau pixels sera donc à une seule dimension et de taille hauteur x largeur

2. Initialisation

Ecrire une fonction d'initialisation

```
image_PBM init_PBM(unsigned char *pixels, int largeur, int hauteur)
```

Si le paramètre pixel est NULL, renvoie une structure image_PBM qui contient la bonne largeur, la bonne hauteur et un espace mémoire réservé de hauteur x largeur rempli de zéro.

Optionnel : Si le paramètre pixel n'est pas NULL, renvoie une structure image_PBM qui contient la bonne largeur, la bonne hauteur et un espace mémoire réservé de hauteur x largeur rempli des éléments se trouvant dans le tableau pixels.

Attention à l'allocation mémoire dans les deux cas !

3. Affichage

Ecrire une fonction qui affiche une image_PBM avec les critères du préambule

```
void affiche_PBM(const imagePBM* image)
```

Note : pour voir les espaces « vides » on peut aussi afficher un point «.».

4. Premiers tests

Créer dans la fonction main un tableau d' **unsigned char** rempli de 0 et de 1 de petite taille avec une hauteur et une largeur définie (4x4 par exemple)

Initialiser une image_PBM puis l'afficher avec **affiche_pbm**.

Une fois terminé, agrandissez votre grille à 10 x 10 puis retestez.

Le tableau suivant contient un « *glider* », une structure qui glisse dans le temps sans mourir.

```
unsigned char glider[100] = {
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
        0, 1, 0, 1, 0, 0, 0, 0, 0, 0,
        0, 0, 1, 1, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};
```

5. get et set

Pour simplifier la modification du tableau de pixels et son accès avec deux variables i et j, nous allons coder deux fonctions : une fonction d'accès get qui renvoie la valeur du pixel placé en i, j :

```
unsigned char get(const imagePBM* image, int i, int j)
```

Puis une fonction set qui écrit le nombre indiqué en paramètre valeur dans la case i, j de la structure :

```
void set (const imagePBM* image, int i, int j, unsigned char valeur)
```

Tester ces deux fonctions en affichant des images et des valeurs de pixel

De cette façon nous accèderons plus facilement aux informations utiles de l'image. Le tableau est aplati en mémoire (plus simple), mais nous y accédons avec deux coordonnées (plus simple).

6. Copie d'image

Le jeu de la vie remplit les cases d'une nouvelle génération de cellules en fonction de la génération précédente. Nous allons donc créer un mécanisme de copie qui va nous permettre de stocker la génération courante pour devenir la précédente. La nouvelle génération sera alors créée dans le tableau courant (par écrasement).

Question à se poser : « que se passerait-il si on ne faisait pas de copie ? »

Créer une fonction **copy_PBM** qui prend en paramètre une **image_PBM** et qui copie tous ses éléments dans une nouvelle **image_PBM** que l'on retournera.

```
imagePBM copy_PBM(const imagePBM* image)
```

Pour tester cette fonction :

Utiliser dans la fonction **main()** l'**image_PBM** créée dans la section 4. Déclarer une deuxième **image_PBM** copie qui sera la copie de la première par appel à **copy_PBM**.

Changer ensuite quelques valeurs dans la copie (avec get et set), par exemple les 4 premiers pixels.

Afficher ensuite les deux images.

La deuxième image doit être la copie exacte de la première, excepté les 4 premiers pixels. Si les quatre premiers pixels sont identiques dans les deux images, votre copie est ratée.

Note : si les quatre premiers pixels sont identiques, ceci veut dire que lorsque vous changez la copie, vous changez aussi l'original, ce n'est pas ce que nous souhaitons. Chaque **image_PBM** *doit occuper son propre espace mémoire.*

7. Cycle de vie

Cette fonction est difficile à écrire et il convient de bien lire la consigne et de faire un brouillon avant de commencer. Ne pas oublier de faire des commentaires.

Ecrire une fonction qui suit le principe de cycle de vie expliqué dans le préambule qui va modifier **une** image_PBM (les pixels seulement).

Compter le nombre de cellules vivantes autour de la cellule (entre 0 et 8)

En fonction de l'état de la cellule, renvoyer 0 ou 1 selon l'état suivant de la cellule.

```
void lifeCycle(imagePBM* generation)
```

Note : selon votre « point de vue » vous pouvez avoir envie de considérer qu'au bord de la grille, il n'y a pas de voisins (donc aucune cellule vivante). Vous pouvez aussi considérer que la grille est « torique », c'est-à-dire que le bord droit et le bord gauche sont virtuellement collés (même chose pour le haut et le bas). Choisissez votre méthode et indiquez votre choix

Conseil : avant de calculer les valeurs par rapport au jeu de la vie, entrainez-vous en faisant se déplacer tous les pixels de l'image d'un pixel vers la droite. Essayez de faire se déplacer une figure de gauche à droite sans décalage vers le haut ou le bas. Quand ce principe fonctionne, essayez de la faire avancer vers le bas.

Ne commencez qu'ensuite à réfléchir à l'algo du cycle de vie qui prend en compte les huit voisins.

Prenez votre temps. Faites des schémas au brouillon sur une feuille.

Test et création du jeu :

Dans une boucle dans la fonction main() utilisez votre **image_PBM** de départ, puis appliquez lui un cycle de vie, affichez la, et nettoyez l'écran avec la fonction clearscreen() ci-dessous.

Votre boucle peut être une boucle for de 5, 10 ou 100 générations (commencez petit)

```
void clearScreen ()
{
  printf ("\x1b[2J");
```

```
  printf ("\e[1;1H\e[2J");
  printf ("\n");
}
```

Note : on trouve sur internet plusieurs implémentations de ***clearscreen****. Aucune ne marche vraiment bien, si vous en trouvez une meilleure, utilisez-la.*

Lancer votre jeu dans un terminal WSL aura un effet nettement meilleur que dans Clion. Ne pas dépasser la taille du terminal pour un affichage optimal (80 environ). Avec une image 10x10 on travaillera plus facilement.

8. Ecrire une image PBM dans un fichier

Nous allons commencer par écrire une **`image_PBM`** dans un fichier pbm. Pour être sûr que son image est bien formatée, vous pourrez utiliser le logiciel libre et gratuit GIMP. C'est un des seuls à lire nativement ce format.

Le format d'un fichier PBM est le suivant

La première ligne contient le mot P1 (et seulement cela) : il indique que l'image est au format noir et blanc, il indique aussi que le fichier n'est pas binaire (nous allons commencer par des choses simples)

Il peut y avoir plusieurs lignes de commentaires, elles doivent toutes commencer par #

La ligne qui suit les commentaires contient la largeur puis la hauteur, séparée par un espace

10 10

Le reste du fichier est une suite de 0 et de 1.

On revient à la ligne tous les 70 caractères.

Note : Pour l'écriture de votre fichier vous pouvez aussi revenir à la ligne chaque fois que la largeur est atteinte pour être capable de le relire vous-même avec un bloc note. GIMP lira aussi votre fichier. Par contre, GIMP respectera les 70 caractères si vous exportez l'image.

Ecrire une fonction **`write_pbm`** qui prend en paramètre un chemin vers un fichier (par exemple « **`test.pbm`** ») et une **`image_PBM`** puis écrit dans le fichier PM au format pbm

```
int write_pbm(const char* filename, const imagePBM* image)
```

*<u>Attention</u> : ce n'est pas parce que vous écrivez « **`.pbm`** » sur l'extension de votre fichier que votre fichier sera forcément un vrai fichier pbm. « L'habit de fait pas le moine », c'est bien la structuration du fichier qui compte.*

<u>Note</u> : avec le logiciel GIMP pour exporter une image pbm il faut bien faire exporter (et pas enregistrer) puis choisir le format ASCII qui nous permettra de lire le fichier avec un bloc note.

Pour tester votre fonction

Prenez votre première **`image_PBM`** de test, puis écrivez la sur un fichier **`test.pbm`**. Puis avec l'explorateur Windows, cherchez votre fichier, et ouvrez-le avec un bloc note. Si l'aspect du fichier correspond à l'attendu, ouvrez l'image avec GIMP. Si votre image correspond à ce que vous vouliez avoir c'est gagné.

9. Lire un fichier PBM

Pour pouvoir commencer avec une image plus grande que notre petit tableau 10x10 et avoir un joli jeu de la vie, il serait intéressant de pouvoir charger une image depuis un fichier. On pourrait ainsi dessiner dessus plutôt que de remplir des 0 et des 1 dans le **`main()`**.

Lire une image suit le même principe que l'algorithme d'écriture. Cependant, l'usage de **`fscanf`** est un peu plus compliqué que **`fprintf`**, il faut penser aux pointeurs. Le formatage est souvent compliqué.

Il est utile de penser à utiliser **`getline`** puis gérer chaque ligne une par une. Ensuite, on peut relire le résultat avec un **`sscanf`**.

Pour le tableau de 0-1, **`fgetc`** peut être un bon choix.

Ecrire une fonction **`read_pbm`** qui prend en paramètre un chemin vers un fichier, un pointeur sur une **`image_PBM`** (qui servira de passage par référence)

```
int read_pbm(const char* filename,  imagePBM* image)
```

Tester votre fonction

Dans la fonction **`main()`**, appeler la fonction **`read_pbm`** avec un fichier « test2.pgm » que vous aurez créé préalablement et vérifié avec votre bloc note. La variable de type **`image_PBM`** doit être déclarée mais par forcément initialisée.

Afficher ensuite l'image avec **`affiche_PBM`**, si elles sont identiques, c'est validé. Tester ensuite avec des images différentes

Regardez bien où le programme a écrit votre fichier dans la section 7, c'est à cet endroit qu'il faut placer votre fichier à lire.

Finaliser votre programme en créant une fonction main qui écrit plusieurs images successives sur le disque à partir du cycle de la vie. *Conseil : commencer par écrire 3 images puis vérifier que tout se passe bien.*

10. Ecrire une animation gif

La commande **`convert`** provenant du package **`imageMagick`** permet de convertir une ou plusieurs images vers d'autres formats. La liste est longue.

Installer `convert` sur WSL Debian

```
apt-cache search imagemagick
```

puis chercher le nom du paquet dans la liste et l'installer

```
sudo apt install imagemagick
```

Convertir les images en gif

Se placer dans le dossier du projet. Chercher ses images (logiquement elles sont dans cmake-build-debug).

```
convert -delay 5 cmake-build-debug/image*.pbm cmake-build-debug/resultat.gif
```

- -delay le temps pour changer d'image dans l'animation
- Les images d'entrée sont sélectionnées avec *.pbm

- Donner un nom à la sortie (resultat.gif par exemple)

Ouvrir le gif sous Windows et admirer l'animation

Attention : l'operateur * lit les fichiers dans l'ordre alphabétique : donc l'animation prendra 0 1 10 11 12 13 … 19 2 20 21 … comme ordre. A vous de trouver une astuce pour éviter ce problème.

11. Pour aller plus loin

Editer une image en noir et blanc sous GIMP et avec un outil crayon, placer des pixels noirs. Exporter l'image en PBM, puis vérifier son format avec un bloc note. Rechercher sur internet des modèles de glider, d'oscillateur, de floraison, de générateur de glider… Faire des tests et modifier le nom des fichiers gif résultats pour les échanger avec la classe.