PROGRAMMATION CONCURRENTE

Game of Life

Raed Abdennadher – Orphée Antoniadis – Steven Liatti

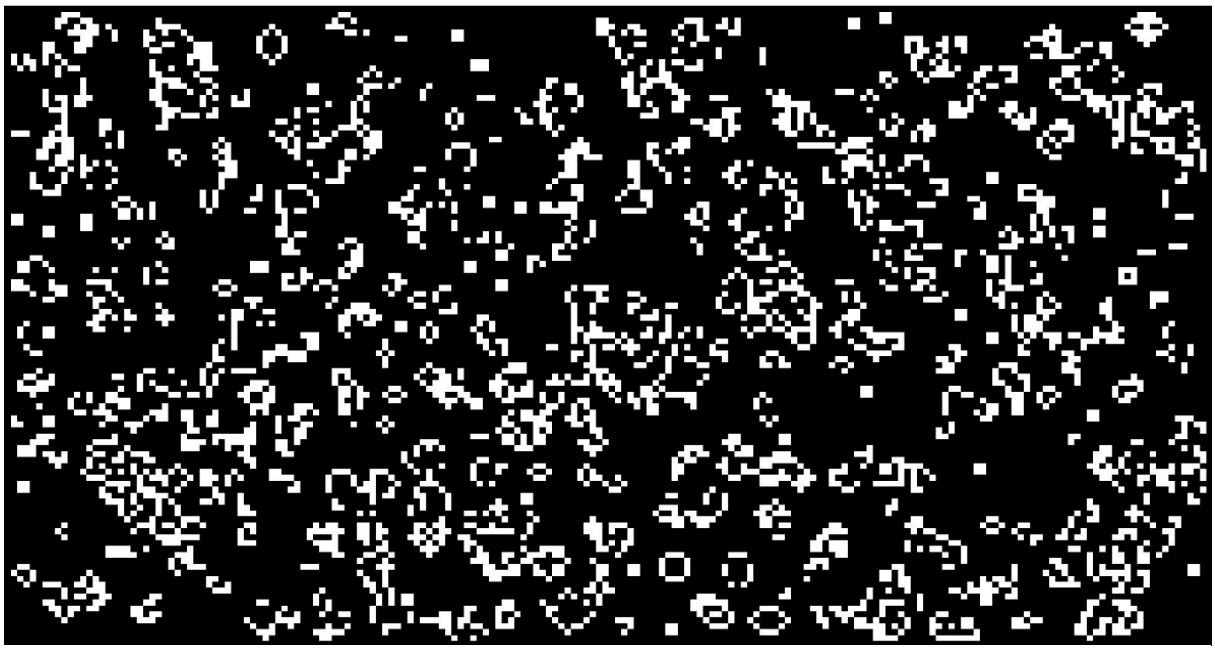


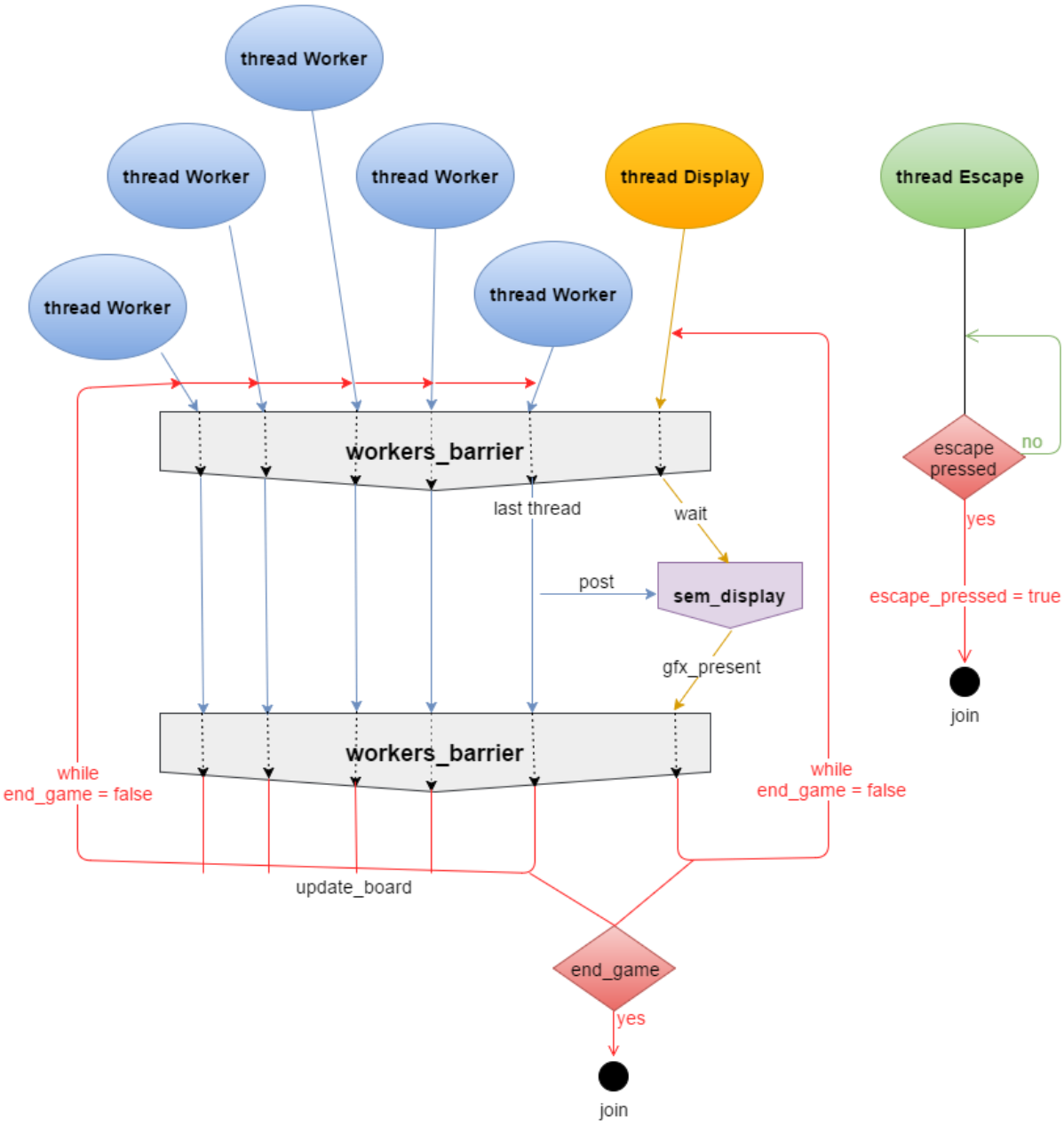
Figure 1 - Capture d'écran du programme en cours d'exécution

Table des matières

Introduction

Le jeu de la vie est une simulation d'une population de cellules qui évolue selon son environnement. Il a été imaginé par John Horton Conway en 1970. La population est représentée par une grille 2D. Chaque cellule est soit morte soit vivante. Les cellules meurent, « renaissent » ou gardent leur état en fonction de plusieurs règles. Pour plus d'infos : https://fr.wikipedia.org/wiki/Jeu_de_la_vie.

Schéma bloc du programme



Méthodologie

Notre programme est divisé en 3 grandes parties :

- La partie gérant les threads travailleurs. Cette partie est subdivisée en deux : d'un côté l'initialisation des *workers* et des structures associées (*workers_management*) et de l'autre les calculs réalisés par les *workers* (*workers_compute*).
- La deuxième partie pour l'affichage (*display board*) qui utilise la librairie graphique *SDL2*.
- Et enfin la partie qui gère la gestion d'interruption du clavier (*keyboard_interrupt*).

Partie 1

L'initialisation des workers et des structures

Notre programme est composé de 4 structures différentes. La première, *cell_t*, représente une cellule de l'écran. Elle contient ses coordonnées dans la matrice de cellules (*width* pour la largeur et *heigth* pour la hauteur), son état présent, son état passé et le nombre de cellules voisines vivantes. La structure *board_t* va contenir la matrice de *cell_t* ainsi que la largeur et la hauteur de la matrice. La structure *sync_t* va simplement contenir toutes nos primitives de synchronisation mais aussi le booléen *escape_pressed* qui va permettre d'indiquer si la touche *esc* a été appuyée, un deuxième booléen *end_game* qui va indiquer si tous nos threads ont fini leur routine et *compute_nb* qui contient le nombre de threads « travailleurs » qui ont fini leur routine. La dernière structure, *worker_t* (qui représente chaque « travailleur ») va contenir les pointeurs sur toutes les autres structures, l'identifiant du *worker*, le nombre total de *workers*, un tableau de pointeurs sur chaque cellule que le *worker* en question va traiter et la taille de ce tableau.

Le fichier *workers_management* contient toutes les fonctions d'initialisation des structures. Nous avons décidé d'allouer dynamiquement nos structures avec la fonction *malloc* afin d'avoir plus de liberté dans la programmation mais aussi pour gagner en lisibilité. Nous avons par conséquent eu à faire une fonction de libération de la mémoire. La fonction *workers_init* va appeler toutes les autres fonctions d'initialisation des structures. Il suffit donc d'appeler cette dernière pour initialiser toutes nos structures. La fonction *update_neighbours* est aussi publique car elle est utilisée après avoir mis à jour l'état des cellules.

Le calcul des threads travailleurs

D'un état à l'autre du jeu, seulement les cellules qui ne se trouvent pas sur les bords sont mises à jour. Les cellules qui se trouvent sur les bords sont à l'état « mortes » et ne sont utilisées que pour le calcul des voisins. Une fois que le thread *display* capture le temps, les travailleurs peuvent commencer leur routine. Chaque travailleur dispose d'un tableau de *cells* qui lui sont assignées. Ce tableau est généré à l'initialisation (*assigned_squares* dans *workers_management.c*) et demeure le même pour chaque *worker* tout au long du programme. Le travailleur met à jour l'état de chaque cellule en fonction de l'environnement de la cellule à l'état passé et des règles du jeu de la vie.

Une fois que tous les travailleurs ont terminé la mise à jour de leurs cellules (grâce au mutex), ils passent la main à l'affichage (au moyen d'un sémaphore) et attendent grâce à la barrière. Une fois l'affichage terminé, les *workers* mettent à jour les voisins et l'état passé de leurs cellules respectives. Cette étape doit obligatoirement être faite après que tous les travailleurs aient mis à jour les cases avec la fonction *update_cell* mais avant que l'état suivant du jeu ait commencé. Finalement, avant de commencer à calculer l'état suivant du jeu, les *workers* attendent à nouveau que l'affichage débute son timer. Ceci est fait en boucle jusqu'à la fin du jeu par l'appui sur la touche ECHAP.

Partie 2

Affichage de la matrice de cellules

Après avoir calculé un nouvel état, l'écran se met à jour en fonction de ce dernier. En fait, l'affichage est relié à deux évènements :

1- Tous les *workers* ont terminé leurs calculs.
2- Le temps d'attente pour la mise à jour est égal à la fréquence entrée par l'utilisateur.

Premièrement, on lance un *timer* pour calculer la durée pendant laquelle les *workers* font leurs calculs. Pour les attendre, on a utilisé une barrière, *workers_barrier,* initialisée avec le nombre de threads travailleurs + 1 (ce +1 vient du fait que le thread d'affichage lui aussi doit terminer son dernier affichage, avant d'afficher le nouvel état). Après, on arrête le *timer*, et on met le thread d'affichage en *sleep(*temps entré par l'utilisateur – temps pour les calculs des *workers)* dans la fonction *adapt_frequency.* Finalement, on affiche le rendu.

Bien évidemment, la fréquence d'affichage maximale sera le temps que tous les *workers* passent pour effectuer leurs calculs.

Partie 3

Gestion d'appui sur la touche « *escape* »

Pour quitter le programme, on a un thread qui vérifie chaque 0,02 secondes (50 Hz) si la touche « *Échap* » est appuyée ou pas. Si c'est le cas, le thread va changer la valeur du booléen *escape_pressed* à *true* et il va quitter sa routine. Et au moment où le thread d'affichage détecte que la valeur de ce dernier est à *true* (à l'aide d'un simple *if*), il va mettre la valeur du booléen *end_game* à *true*. Et puisque tous les threads tournent dans une boucle tant que *end_game* est à *false*, ils vont tous quitter leurs routines dès que la valeur de *end_game* passe à *true*.

Répartition du travail

Steven s'est principalement occupé du calcul des threads travailleurs (*workers_compute* et *workers_management*).

Orphée s'est principalement occupé de l'initialisation et la destruction de toutes les structures et threads, de la gestion des workers (*workers_management*) et du *main* (*gameoflife.c*).

Raed s'est principalement occupé de l'affichage (*display_board*) en utilisant la librairie *gfx* et de l'interruption au clavier (*keyboard_interrupt*).

Pour le reste, et surtout pour l'implémentation et la gestion des mécanismes de synchronisation, nous avons collaboré soit par combinaisons de binômes selon les disponibilités de chacun, soit tous ensemble, essentiellement en mettant nos idées sur papier puis en vérifiant la bonne exécution avec le compilateur.

Problèmes et difficultés

Nous nous sommes pris à deux reprises pour obtenir une répartition efficace et équitable entre les *workers*. La capture de l'appui sur la touche ECHAP n'a pas été évident tout de suite non plus. Mais c'est surtout l'association de tous les mécanismes de synchronisation pour obtenir la séquence voulue qui a été difficile.

Néanmoins, nous avons surmonté ces différents problèmes et avons une version fonctionnelle et sans bugs connus du programme.