ECE
ÉCOLE D'INGÉNIEURS
ENGINEERING SCHOOL

ING 4
Groupe 2

RAPPORT DE PROJET

[Jeu de la vie embarqué]

Le but de ce projet est de réaliser en C, un programme reproduisant le système du jeu de la vie. Tout en respectant les contraintes du sysèmes embarqués.
Ce projet s'inscrit dans notre majeur systèmes embarqués en 4ème année à l'ECE Paris dans notre cours "Advanced C Programming"

Nous attestons que ce travail est original,
qu'il est le fruit d'un travail commun au trinôme et qu'il a été rédigé de manière autonome.

Tristan JEAN Marius LEPERE Eugène LAMBERT

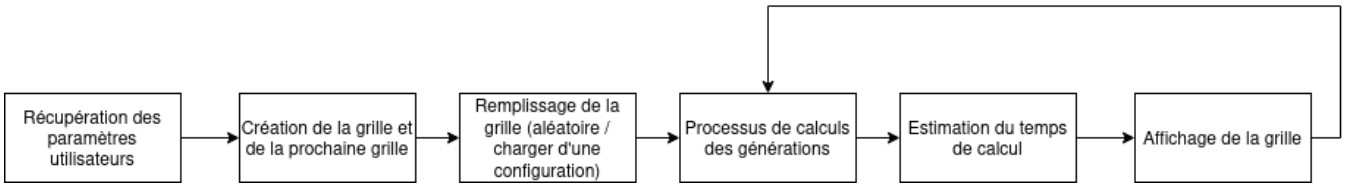
November 30, 2025

Contents

1 Conception

1.1 Struture

Notre code se structure en plusieurs parties :



La première étape est la récupérations des données de l'utulisateur, des paramètres sont passés lors de l'appel du programme tel que, la largeur (width), la hauteur (height), le nombres de générations (gens), la façon dont géré les bordures (boundary), le fichier de configuration (in), le fichier d'enregistrement (out) et enfin la fréquence choisi (target_hz).

1.2 Développement

Afin d'avoir une grille propre et optimisé nous créons une structure grid contenant les attributs :

- width, height : largeur, hauteur
- words_per_row : nombres de valeurs que l'on peut stocké sur une ligne
- data : contenu de la grille
- mode : mode de jeu

On définit le nombre de caractères nécessaires pour stocker une ligne par :

$$words\_per\_row = \left\lceil \frac{width}{W} \right\rceil$$

où *width* est la largeur de la grille en bits, et W est la taille d'un mot (généralement 64 bits). Cela correspond au nombre de blocs de 64 bits nécessaires pour représenter une ligne.
Comme notre strucutre n'est composé que de uint64_t cela nous permet d'optimiser notre espace mémoire.

La génération des tours se fait sur le système du jeu de la vie, on observe les voisins autour de notre cellule et on estime si la cellule doit vivre ou mourir. On stocke la prochaine valeur de l'emplacement de la cellule dans la prochaine grille *Next*. Qui sera donc par la suite la grille courante jusqu'à la dernière génération demandé par l'utilisateur.
Pour charger une configuration on lit dans le fichier texte ce qui va nous permettre de remplir notre grille automatiquement, on prends donc les 64 premières caractères qu'on va tranformer en *0* pour une cellule **morte** ou *1* pour une cellule **vivante**. Dans l'autre cas, si l'on souhaite sauvergarder, nous allons utiliser la largeur et la hauteur de notre grille afin de récupérer les bonnes valeurs et d'obtenir un résultat propre et lisible.

Calcul d'une nouvelle génération :
Pour chaque cellule (x, y) de la grille on utilise le **bit-packing** :
Chaque cellule (x, y) est stockée dans un *word* de 64 bits pour économiser de la mémoire.

1. **Identifier le mot contenant la cellule :** Comme chaque word contient 64 cellules, l'indice du word dans la grille est :

   $$word\_index = y \cdot words\_per\_row + (x \gg 6)$$

   Ici, $x \gg 6$ correspond à $x/64$, c'est-à-dire dans quel word de la ligne se trouve la cellule.

2. **Identifier le bit correspondant :** La position de la cellule dans ce word est :

$$bit\_index = x\&63$$

L'opération & 63 équivaut à $x \underline{m} odulo 64$ et donne quel bit du word correspond à la cellule.

3. **Créer un masque pour isoler la cellule :** On construit un *mask* qui a un 1 uniquement à la position du bit de la cellule :

$$mask = 1 \ll bit\_index$$

Ce masque permet de lire ou modifier uniquement ce bit.

4. **Lecture de la cellule :** Pour savoir si la cellule est vivante ou morte :

$$tat = (word\&mask) \neq 0$$

5. **Écriture de la cellule :** Pour mettre la cellule vivante :

$$word| = mask$$

Pour la mettre morte :

$$word\& = \sim mask$$

Estimation et contrôle du temps de simulation :
La simulation mesure le temps nécessaire pour calculer chaque génération et ajuste l'affichage pour respecter une cadence cible (`target_hz`). Pour chaque génération, on mesure le temps CPU nécessaire à la fonction `next_generation` via `clock_gettime(CLOCK_MONOTONIC)`. La durée de calcul par génération est la soustraction entre le timer à la fin de la génération et celui au début de la génération.

Après le calcul, la simulation attend le temps restant pour atteindre la cadence exacte, en utilisant `nanosleep` si nécessaire.

On mesure le temps complet d'une frame (calcul + attente)
Après toutes les générations, on calcule :

- le temps moyen par frame
- le pire temps observé
- le jitter : Soustraction entre le pire temps et le meilleur temps

Pour optimiser le code minimise donc l'utilisation du CPU en dormant quand le temps restant avant la prochaine frame est suffisant, garantissant une cadence stable même si le calcul est rapide.

Affichage de la grille :
Enfin, pour afficher la grille on récupère toutes les cellules de notre grille et affichons *'X'* si celle-ci est vivante et *'-'* dans l'autre cas si elle est morte

2 Résultat de mesure de temps

Sur 1000 générations et une grille de 100x100 voici les résultats obtenus pour un total de 1000 générations.

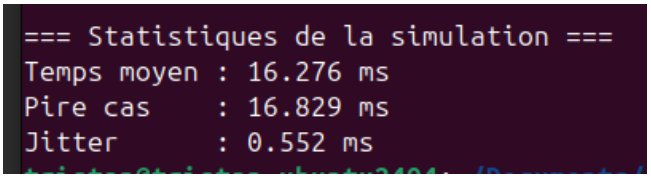


Figure 1: Temps de mesure pour 1000 générations sur une grille 100x100 générée aléatoirement

Nous pourrions donc nous demander : comment se fait-il que notre code n'atteint pas exactement les 60 Hz, soit 16.667 ms.
Cela s'explique car la fonction *printf*, qui nous sert à afficher notre grille, prend un petit peu de temps, rendant notre résultat imprécis. En revanche, si on enlève notre affichage, voici ce que cela donne :

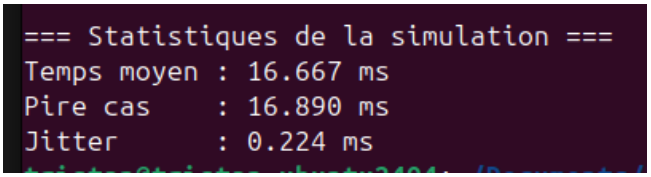


Figure 2: Temps de mesure pour 1000 générations sur une grille 100x100 générée aléatoirement sans printf

On observe ici une mesure parfaite car 60hz = 16.667ms de même si on passe à 120hz le résultat est le suivant :

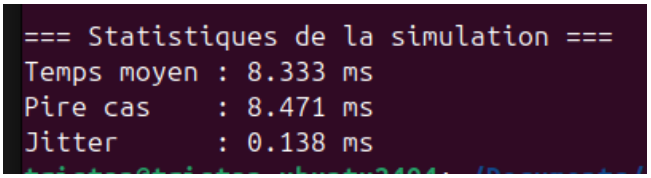


Figure 3: Estimation du temps ave une target_hz à 120hz

On obtient encore **le résultat attendu**.

3 Réponses aux questions

Question 1 : Comment détecter automatiquement un oscillateur dans votre programme ? Pouvez-vous concevoir un moyen de démontrer qu'un motif est entré dans un cycle ?
Un oscillateur est un motif qui se répète, des motifs stables dans le jeu de la vie existent :
Exemple de motifs stable :

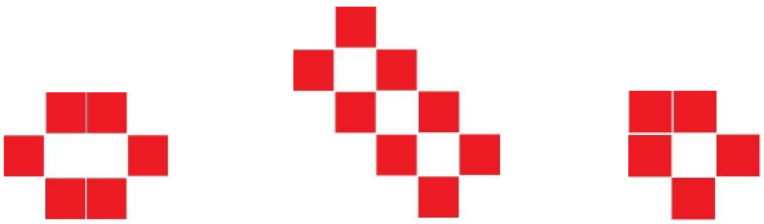


Figure 4: Motifs stables

On observe donc ici que le motif de répète, afin d'identifier un motif répétitif nous pourrions enregistrer les grilles précédentes afin d'observer si sur des zones les cellules sont dans le même état qu'une grille précédente, ainsi il serait possible d'identifier des schémas et donc des oscillateurs.

Questions 2 : Que doit-il se passer si l'utilisateur demande une grille trop grande par rapport à la mémoire disponible ? Le programme doit-il s'arrêter proprement avec un message clair, refuser l'entrée ou proposer une réduction automatique de la taille ?

Étant donné une limite stricte de mémoire de 64 KiB pour le monde 2D, il faut d'abord calculer le nombre de bits nécessaires pour stocker la grille :

$$bits\_nécessaires = width \times height$$

$$octets\_nécessaires = \frac{bits\_nécessaires}{8}$$

Puisque 64 KiB = 65 536 octets, on doit vérifier :

$$octets\_nécessaires \leq 65\,536$$

Pour gérer une grille trop grande il fraudrait donc :

- **Option 1 : arrêt propre** – le programme refuse la création de la grille et affiche un message clair à l'utilisateur.
- **Option 2 : réduction automatique** – le programme ajuste la largeur et la hauteur pour que la grille tienne dans la mémoire disponible.
- **Option 3 : saisie répétée** – l'utilisateur est invité à entrer de nouvelles dimensions.

Cette vérification permettrais d'éviter les débordements mémoire et les comportements indéfinis lors de la création de grilles trop grandes.

Question 3 : Comment un planeur (glider) se comporterait-il dans chacun de ces cas ?
Exemple de planeur :

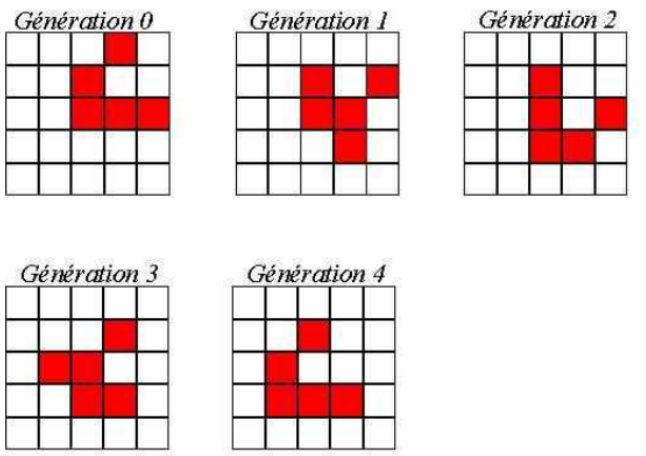


Figure 5: Planeurs motifs

Le planeur se déplace diagonalement à travers la grille en répétant un motif cyclique. Si la grille utilise des bords de type **Edge**, tout ce qui sort de la grille est considéré comme mort, ce qui signifie que le planeur finira par se heurter à un bord et sera détruit après quelques générations. Avec un bord **Toroidal** ou tore, la grille s'enroule sur elle-même : le planeur qui atteint le bord droit réapparaît immédiatement sur le bord gauche, et de même pour le haut et le bas, permettant au planeur de se déplacer indéfiniment sans interruption. Dans le cas d'un bord **Mirror**, les coordonnées au-delà de la grille se reflètent à l'intérieur, de sorte que le planeur se heurtera virtuellement à son reflet, modifiant sa trajectoire et pouvant perturber son mouvement régulier. Enfin, avec une **Alive rim**, toutes les cellules en dehors de la grille sont considérées comme vivantes, ce qui aura pour effet de bloquer ou de transformer le planeur lorsqu'il atteint les bords, provoquant souvent la destruction ou la modification

du motif initial. Ainsi, le comportement d'un planeur dépend fortement de la façon dont les bords sont traités, allant de la destruction rapide à un déplacement infini ou à des perturbations de sa trajectoire.

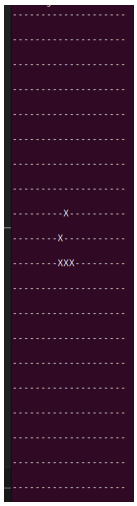


Figure 6: Glider 32x32

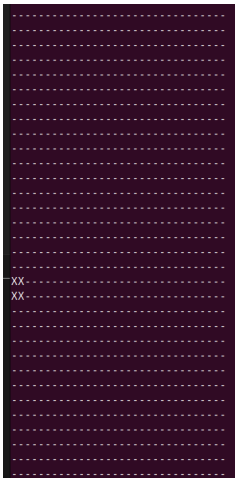


Figure 7: Bordure edge

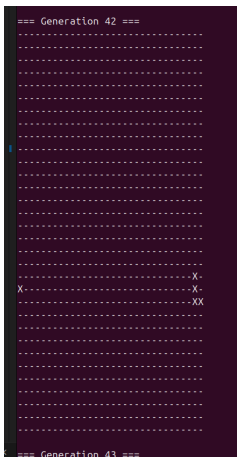


Figure 8: Bordure torus

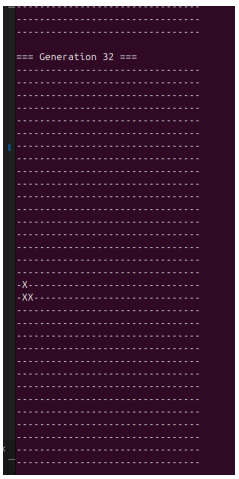


Figure 9: Bordure mirror

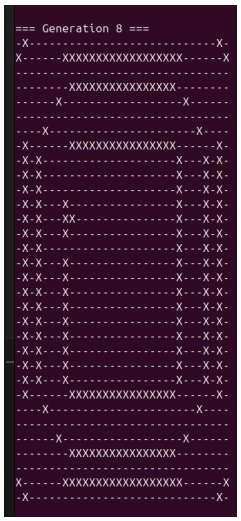


Figure 10: Bordure rim

Cette observation concernant les gliders nous permet de confirmer dans un premier temps que la génération de notre projet marche mais aussi que notre gestion des différentes bordures fonctionnent. En effet, pour la bordure **edge** le glider s'arrête au moment où il touche le mur et devient un motif stable. Pour la bordure **torus** elle va bien de l'autre côté de la grille, pour la **mirror** deux gliders se touchent amenant le glider dans un état stable enfin la bordure **rim** nous montre que le glider rentre en collision avec les cellules qui sont partis de l'extérieur cosant une collision.

4 Résultat du projet

Notre projet fonctionne dans son intégralité. En effet toutes les étapes sont correctement implémenté et permettent le bon fonctionnement du jeu et dans la limite du temps demandé.
Voici une illustration complète d'un déroulé du jeu :

```
tristan@tristan-ubuntu2404:~/Documents/ECE/Advance_C/Projet/Jeu_Vie$ ./output --width 100 --height 100  --gens 100 --bounda
ry torrus --in glider.txt --out end.txt --target-hz 60
```

Figure 11: Lancement du code avec paramètres

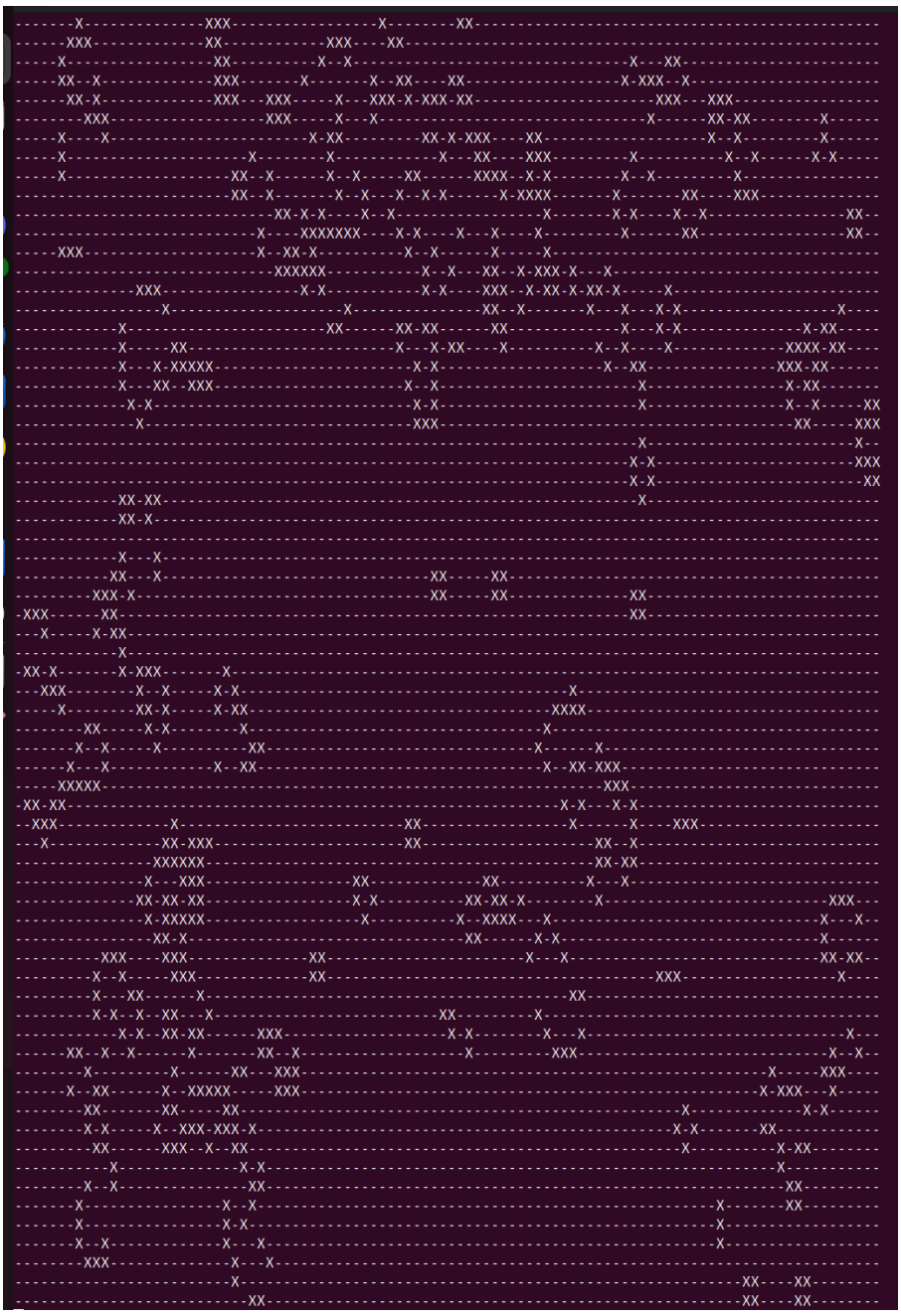


Figure 12: Execution du code avec affichage ou sans affichage (100x100)

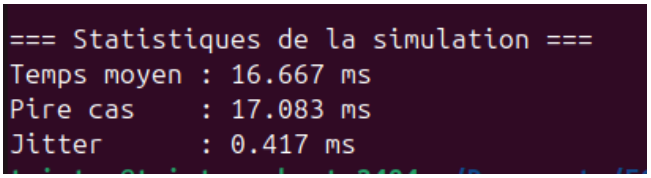


Figure 13: Estimation du temps de calcul

Nous avons donc, après différents tests pu observer que notre projet fonctionnait dans son intégralité respectant le cahier des charges.

5 Makefile

Afin d'obtenir un moyen plus simple de compiler et d'utiliser un fichier makefile afin d'éxécuter la commande de compilation en utilisant simplement le mot clé **make**
Ansi notre makefile est capable d'éxécuter la commande suivante :

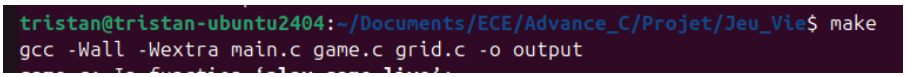


Figure 14

6 Réflexion sur le projet

Réflexion d'ingénieur embarqué

Ce projet nous a permis de réfléchir comme des ingénieurs embarqués, en prenant en compte les contraintes spécifiques des systèmes à ressources limitées. Travailler sur le *Jeu de la Vie* nous a confrontés à des choix fondamentaux sur la manière de représenter et de traiter les données efficacement. En particulier, nous avons compris l'importance de l'optimisation mémoire. Sur une grille de grande taille, il devient rapidement impossible de stocker chaque cellule de manière naïve. Cette contrainte nous a poussés à envisager des représentations plus compactes, comme l'utilisation de bits individuels pour chaque cellule, et à réfléchir à la structure des données afin de minimiser l'utilisation de la mémoire sans sacrifier la clarté du code.
De plus, la réflexion sur le temps de calcul nous a appris à ne pas considérer chaque opération de manière isolée. En systèmes embarqués, chaque boucle et chaque condition peut avoir un impact significatif sur les performances globales. Nous avons ainsi été amenés à penser à des stratégies pour réduire le nombre de calculs inutiles, comme se concentrer sur les cellules susceptibles de changer d'état ou optimiser le parcours de la grille.
Enfin, ce projet nous a fait réaliser que penser comme des ingénieurs embarqués ne consiste pas seulement à coder : il s'agit d'anticiper les limitations matérielles, de prendre des décisions conscientes sur la gestion des ressources et de concevoir des solutions pragmatiques. Cette approche influence non seulement l'efficacité du programme, mais aussi la qualité globale du design logiciel dans un contexte embarqué.

7 Annexes

Le code disponible sur **Github :** https://github.com/Tristoun/Jeu_de_la_vie_C