Le Jeu de la Vie

Joudy BENKADDOUR, Paul-Antoine CHARBIT, Fabien CASTELLINI

Présentation du sujet

Nous connaissions déjà le Jeu de la Vie avant de choisir le sujet, chacun de différente manière, mais nous pensons que la meilleur reste de regarder une petite vidéo YouTube à ce sujet, même si chacun des parties ici (nous 3 et vous) connait le jeu, nous allons le présenter. Le Jeu de la Vie est crée par le mathématicien John Horton Conway en 1970. C'est un jeu à zéro joueur, ce qui signifie que son évolution et donc le résultat est déterminée par son état initial (ce qui est un peu dommage mais bon s'il y avait de l'aléatoire, le jeu n'aurait plus aucun sens). Le Jeu de la Vie est intéressant car il démontre comment des règles simples peuvent conduire à des comportements complexes.

On peut aussi tester quelques structures comme les structures stable ou les périodiques.

Mais ! Ce qui est vraiment intéressant avec ce jeu, c'est lorsque l'on découvre les canons et tout ce que l'on peut faire avec.

Explication des mathématiques permettant de répondre au sujet

Il n'y a pas beaucoup de mathématiques ici, certes il y a des 1 et des 0. La seul opération est l'addition et le chiffre le plus grand que l'on peut obtenir est 8 comme nombre de voisin.

Le Jeu de la Vie est simple, il y a quatre règles qui déterminent si une cellule sur une matrice sera vivante ou morte dans la génération suivante. Ces règles, qui reposent sur le nombre de voisins vivants d'une cellule :

Naissance : Une cellule morte avec exactement trois voisins vivants devient vivante.

Survie : Une cellule vivante avec deux ou trois voisins vivants reste vivante.

Décès par sous-population : Une cellule vivante avec moins de deux voisins vivants meurt.

Décès par surpopulation : Une cellule vivante avec plus de trois voisins vivants meurt également.

Présentation du code clairement commenté

Comme c'est notre première fois sur C++, nous avons essayé de commenter au maximum notre code. Le plus dur est de s'habituer à la syntaxe, heureusement le projet en tant que tel ne demande pas de compétence avancée en C++. Donc nous avons 4 fonctions dans notre projet :

Le main() pour lancer le jeu et initialiser notre base (on aurait pu faire une fonction à cet effet). :

```
int main() {
    // Création de notre grille
    vector<vector<int>> grille(HAUTEUR, vector<int>(LARGEUR, 0));

    // Initialisation de la grille avec un motif plus grand
    grille[15][40] = 1;
    grille[15][41] = 1;


    // Efface l'écran (pareil ça marche mais... grâce à une volonté qui nous
dépasse)
    cout << "\033[2J\033[H";

    while (true) {
        affichageGrille(grille);
        grille = majGrille(grille);

        // Met en pause le code pour bien voir "l'animation"
        this_thread::sleep_for(chrono::milliseconds(100));
    }

    return 0;
}
```

majGrille(const vector<vector<int>>& ancGrille) qui permet donc de mettre à jour notre grille, donc de passer au pas de temps suivant. Elle prend en argument une grille et en renvoie une autre

```
// Fonction pour mettre à jour la grille en appliquant les règles du Jeu de la
Vie
// Cette fonction prend en entrée la grille actuelle et renvoie une nouvelle
grille mise à jour.
vector<vector<int>> majGrille(const vector<vector<int>>& ancGrille) {
    // Création d'une nouvelle grille vide pour la prochaine génération.
    vector<vector<int>> nvGrille(HAUTEUR, vector<int>(LARGEUR, 0));

    // Parcours de chaque cellule de la grille
    for (int i = 0; i < HAUTEUR; ++i) {
        for (int j = 0; j < LARGEUR; ++j) {
            // Compte les voisins vivants de la cellule courante
            int voisinVivant = nbVoisinVivants(ancGrille, i, j);

            // Applique les règles du Jeu de la Vie :
            if (ancGrille[i][j] == 1 && (voisinVivant == 2 || voisinVivant == 3))
{
                // Règle de survie : Une cellule vivante avec deux ou trois
voisins vivants reste vivante.
                nvGrille[i][j] = 1;
            }
            else if (ancGrille[i][j] == 0 && voisinVivant == 3) {
                // Règle de naissance : Une cellule morte avec exactement trois
voisins vivants devient vivante.
                nvGrille[i][j] = 1;
            }
            else {
                // Règle de mort : Dans tous les autres cas, une cellule meurt ou
reste morte.
                nvGrille[i][j] = 0;
            }
            // La cellule est mise à jour dans la nouvelle grille en fonction des
règles ci-dessus.
        }
    }
    // Retourne la nouvelle grille avec la prochaine génération de cellules.
    return nvGrille;
```

nbVoisinVivants(const vector<vector<int>>& grille, int x, int y) qui permet simplement de compter le nombre de voisin en vie autour d'une cellule.

```
// Fonction pour compter les voisins vivants d'une cellule dans la grille donc
renvoie un integer
// La fonction prend en entrée la grille et les coordonnées (x, y) de la cellule
de départ. Prend des integers en input
int nbVoisinVivants(const vector<vector<int>>& grille, int x, int y) {

    // Initialisation du nombre de voisins vivants
    int voisinVivant = 0;

    // On parcourt les cellules autour de la cellule de départ
    // D'abord celle sur l'axe vertical
    for (int i = -1; i <= 1; ++i) {
        // Puis sur l'axe horizontal
        for (int j = -1; j <= 1; ++j) {
            // On ne compte la i=0 et j=0 car c'est la cellule de départ
            if (i == 0 && j == 0)
                continue; // Continue passe à la prochaine itération de la boucle

            // Coordonnées des cellules voisines
            int nvX = x + i;
            int nvY = y + j;

            // Si le voisin est à l'intérieur de la grille
            if (nvX >= 0 && nvX < HAUTEUR && nvY >= 0 && nvY < LARGEUR) {
                // Incrémentation du compteur si un voisin est vivant
                voisinVivant += grille[nvX][nvY];
            }
            // Si le voisin est en dehors de la grille, il est considéré comme
mort
        }
    }
    // Retourne le nombre total de voisins vivants autour de la cellule de départ
    return voisinVivant;
}
```

affichageGrille(const vector<vector<int>>& grille) qui permet d’afficher la grille.

```
// Fonction pour afficher la grille avec un void car on n'attend pas de valeur en
retour
// On a 2 fois vector car on affiche une matrice de nombre entier (même binaire)
// La fonction prend en argument notre grille qui est une matrice 2D et on met
const car on ne s'attend pas
// à ce que la fonction change la variable
void affichageGrille(const vector<vector<int>>& grille) {
    // Créer une "illusion" d'animation. (On ne sait pas comment ça marche
mais... ça marche)
    cout << "\033[H";
    // On passe d'abord sur l'axe vertical
    for (int i = 0; i < HAUTEUR; ++i) {
        // Puis l'axe horizontal
        for (int j = 0; j < LARGEUR; ++j) {
            // On vérifie l'état de la cellule : 1 (vivante) ou 0 (morte)
            if (grille[i][j] == 1)
                // Astérisque pour une cellule vivante
                cout << "*";
            else
                // Espace pour une cellule morte
                cout << " ";
        }
        // Ligne suivante
        cout << "\n";
    }
}
```

Problèmes rencontrés et la manière de les surmonter

Alors tout d'abord la première difficulté est de coder en C++, cela reste très différents des langages que l'on a appris jusqu'à présent (Python, VBA…) notamment par la syntaxe exigeante, toujours typer les variables, les parenthèses, les accolades…

Ensuite, en petite difficulté, il y a eu la manipulation des matrices que nous n'avons pas vu en cours (uniquement les vecteurs) mais cela se rapproche tout de même de ce que l'on a connu.

Nous trouvons aussi que le débogage sur ce projet était un peu compliqué, car nous devions voir et comprendre comment réagissait notre code alors qu'il est censé être dynamique.

Aussi les codes d'erreurs ne sont pas très clairs, il est parfois difficile de comprendre quelle est notre erreur.

Focus particulier sur une difficulté rencontrée

LA difficulté de ce projet est l'affichage, cela a pris 2x plus de temps que de coder le jeu. Nous avons vraiment voulu voir l'évolution de notre figure (c'est le principal intérêt du jeu, son artistique).

Au départ, il fallait bien ajuster notre dimension de grille, histoire de bien voir l'évolution de notre figure, que le jeu est la liberté de s'exprimer, ensuite nous avons dû choisir les bonnes formes pour représenter notre figure (On a essayé de garder les 0 et les 1 mais c'était illisible).

Ensuite en lançant le code, on voit nos figures défiler à la vitesse de la lumière, il a donc fallu mettre un sleep pour temporiser afin de suivre l'évolution.

Autre problème sur le terminal, on voyait notre jeu défiler vers le bas et donc cela nuit à l'expérience utilisateur pour suivre l'évolution du jeu, on a donc essayé de faire en sorte de n'avoir qu'une seule grille et de suivre son évolution au fil du temps, mais impossible.

On s'est donc dit qu'il fallait ruser, et effacer le terminal avant d'afficher notre nouvelle figure mais on avait encore un décalage sur notre grille qui « glissait » vers le bas. On a essayé de réadapter les dimensions de nos figures mais rien n'y fait.

Et donc il s'avère que des codes « ANSI » qui font de la magie sur notre terminal pour faire apparaitre notre figure parfaitement sur le terminal, nous n'en comprenons nullement la signification mais… ça marche !

Cela nous permet d'avoir une « fausse » animation où l'on voit bien notre grille évoluer au fil du temps.

Conclusion et perspectives

Ce projet a permis de découvrir C++ et d'enfin avoir « notre » jeu de la Vie, c'est un bon projet de départ (sauf pour l'affichage mais le jeu perdrait encore son intérêt). On s'est amusé avec quelques figures (et un beau canon). Je pense qu'il serait drôle de voir ce que donnerait le jeu en 3D avec une belle visualisation mais pas sur un terminal. Il doit surement exister des interfaces plus adaptés pour cela.