
Projet C++ : Game Of Life



Etudiants :
BERBAGUI Amine
HACHANI Ghassen

Enseignante :
Mme BRAUNSTEIN Cécile

MAIN 4 2021-2022

30 décembre 2021

Table des matières

1	Introduction	2
2	Fonctionnement du jeu	3
3	Résultat du jeu	4
4	Présentation des contraintes utilisées	5
5	Diagramme UML de notre jeu	6
6	Parties dont nous sommes fiers !	7
6.1	Le bruit de Perlin	7
6.2	Plot de la simulation	8
6.3	La partie audio	9
7	Conclusion	10

1 Introduction

Dans le cadre de notre projet en **C++**, nous avons comme objectif de créer une application respectant une liste de contraintes (cf **sujet**) parmi lesquelles le thème de l'application qui est *There is no planet B* donc un thème lié à la nature et l'écologie.

Après plusieurs changement de choix, nous nous sommes finalement orientés vers un jeu comportant une interface graphique qui simulerait un environnement animal, une sorte d'écosystème où cohabiterait deux espèces animales : **les loups** et les **moutons**. Voici le lien de la vidéo dont nous nous sommes inspirés.

Coding Adventure: Simulating an Ecosystem

La partie graphique du jeu se fera grâce à la librairie **SFML** qui est une librairie assez simple d'utilisation et complète. Nous l'avons choisis car son mode d'utilisation est similaire à la librairie **SDL**.

Nous avons comme idée aussi de présenter les statistiques de la simulation en temps réel sur la fenêtre graphique ainsi que quelques options que l'utilisateur peut utiliser directement comme :

- La possibilité d'accélérer la vitesse de la simulation
- La possibilité de ralentir la vitesse de la simulation
- La possibilité de mettre la simulation sur pause
- La possibilité de cacher les statistiques.

Vous pouvez par ailleurs accéder au dépôt git du projet pour plus d'informations notamment le README qui détail la compilation du projet : [Git](#)

2 Fonctionnement du jeu

Passons maintenant au fonctionnement du jeu dans les détails. Comme expliqué en introduction, notre simulation se fera entre deux espèces : les **loups**, représentés sur la fenêtre par des cercles bleus ainsi que les **moutons** représentés par des cercles oranges. La nourriture, elle, est représentée en blanc. La simulation commence avec 30 moutons et 8 loups.

Ces chiffres ont été choisis pour ne pas avoir une simulation trop longue mais ces paramètres peuvent être modifiés dans le fichier `constantes.hpp` du dossier `src`.

À chaque régénération de la fenêtre, les animaux se déplacent pseudo-aléatoirement en utilisant le bruit de Perlin (cf [Partie Perlin](#)). Leurs points de vie diminuent d'un point par déplacement.

Ainsi au cours de la simulation, deux entités interagissent entre elles lorsque les cercles qui les représentent se superpose.

Voici toute les interactions possible :

- Quand un mouton rencontre de la nourriture, le mouton mange la nourriture. La nourriture est régénérée à un autre endroit de la fenêtre et le mouton régénère ses points de vie à la valeur de départ.
- Quand un loup rencontre un mouton, le loup mange le mouton. Le loup régénère ses points de vie, et le mouton disparaît.
- Quand un mouton rencontre un autre mouton, un nouveau mouton naît avec une certaine probabilité (définie à **0.3**).
- Quand un loup rencontre un autre loup, un nouveau loup naît avec une certaine probabilité (définie à **0.3**).
- Quand un animal n'a plus de points de vie, il meurt et disparaît de la fenêtre.

3 Résultat du jeu



FIGURE 1 – Fenêtre de lancement du jeu

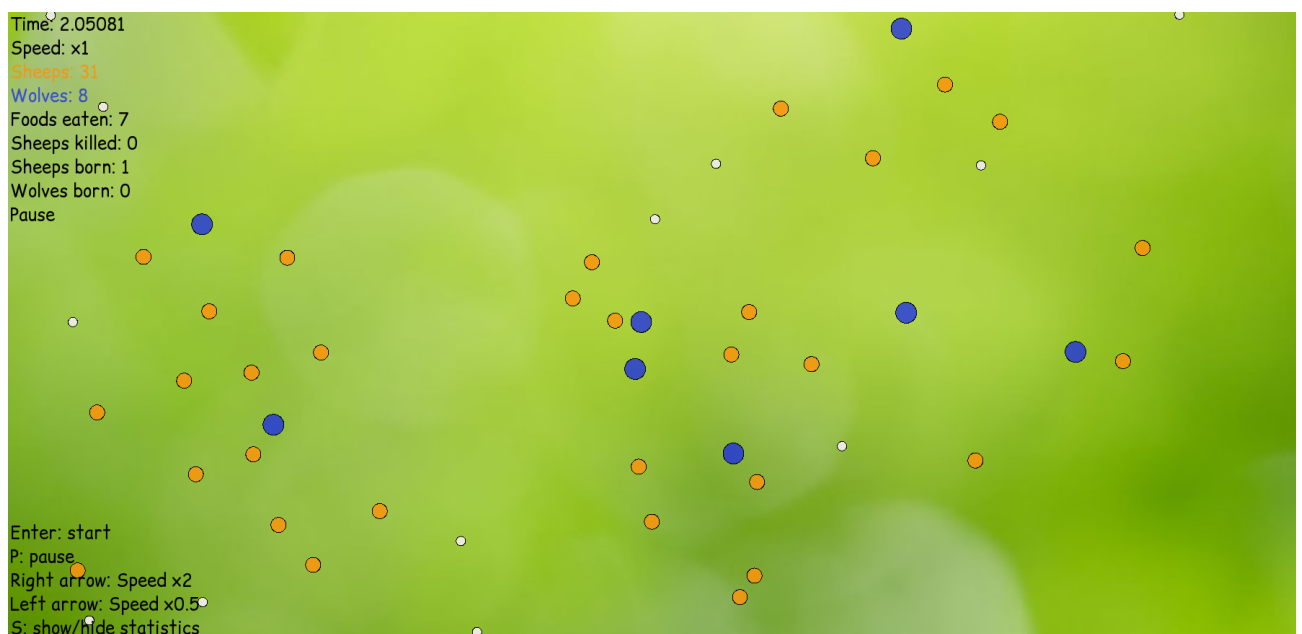


FIGURE 2 – Fenêtre du jeu en cours

4 Présentation des contraintes utilisées

Comme annoncé en introduction, notre projet devait respecter une liste de contraintes que nous devons respecter. Voici les contraintes imposées et la manière dont nous l'avons utilisé :

1. **Avoir 8 classes** : Notre jeu en comporte 11 (avec `sf::CircleShape` et `sf::Drawable`).

2. **3 niveaux de hiérarchie** : $(\text{Sheep}, \text{Wolf}) \longrightarrow \text{Animal} \longrightarrow \text{CircleShape}$.

3. **2 fonctions virtuelles** : On a cette fonction présente dans *animals.hpp*, *foods.hpp* et *plot.hpp*. Ces trois classes héritent de *Drawable* donc ceci permet de les redéfinir à chaque fois pour dessiner les animaux, la nourriture et le plot.

```
virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const;
```

4. **2 surcharges d'opérateurs** : deux surcharges de l'opérateur `+` de type `bool` permettant de gérer la reproduction entre deux loups et deux moutons.

```
bool operator+(Sheep &sheep) const;  
bool operator+(Wolf &wolf) const;
```

5. **2 conteneurs STL** : On a une multitude de template dans *Animals.hpp* pour implémenter toutes les méthodes de cette classe comme :

```
template<typename T>  
void Animals<T>::add(const unsigned int nbAnimals)  
  
template<typename T>  
unsigned int Animals<T>::getNb() const
```

6. **Diagramme UML complet**

cf [Diagramme UML](#)

7. **Code bien commenté**

cf [README](#)

8. **Utilisation d'un Makefile**

cf [Makefile Jeu](#)

cf [Makefile Tests](#)

9. **Utilisation tests unitaires**

cf [Tests](#)

5 Diagramme UML de notre jeu

Notre diagramme étant assez dur à voir, je vous invite à le visualiser de plus près depuis notre dépôt github ici : [UML](#)

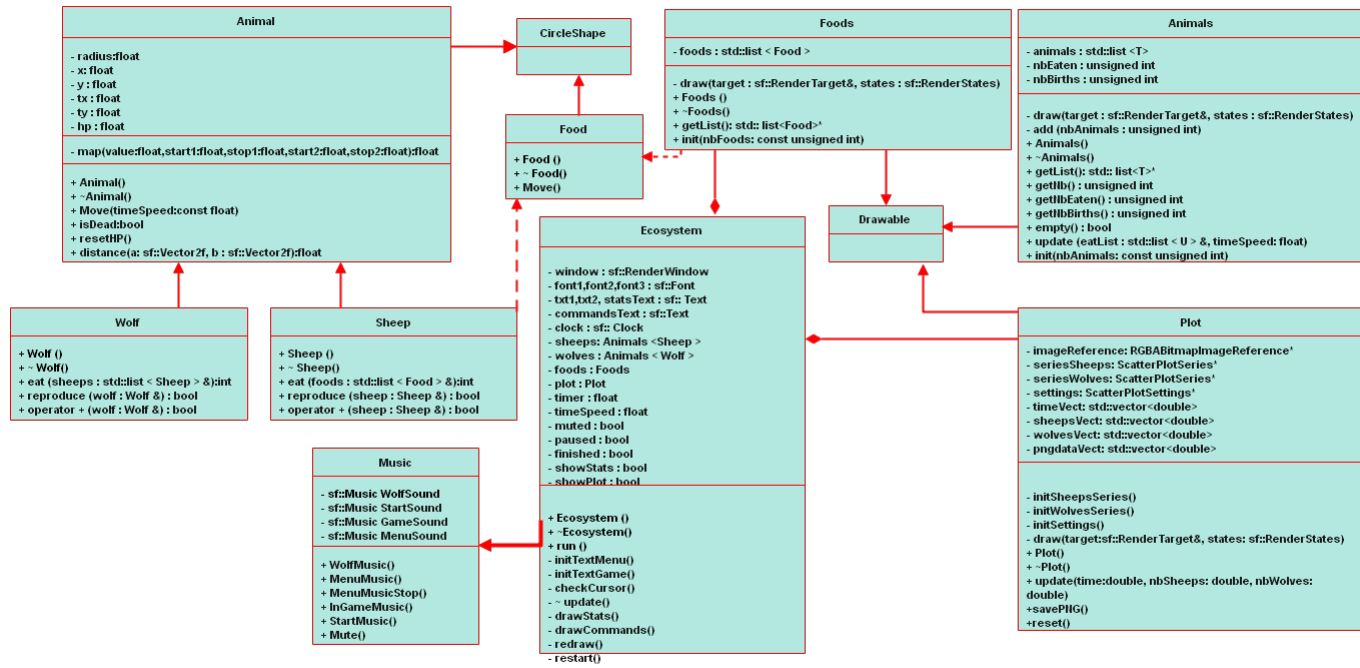


FIGURE 3 – Diagramme UML du jeu

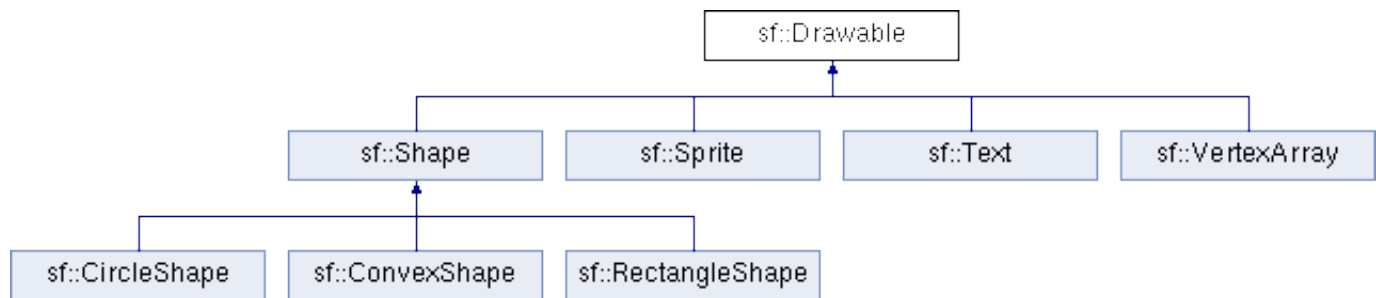


FIGURE 4 – Diagramme d'héritage de la classe Drawable que nous utilisons

6 Parties dont nous sommes fières !

6.1 Le bruit de Perlin

Pour simuler une trajectoire proche de la réalité lors de notre jeu, nous avons implémenté ce qu'on appelle le bruit de Perlin. Derrière ce bruit se cache en réalité un algorithme extrêmement puissant créé par [Ken Perlin](#), qui est souvent utilisé dans la génération de contenu procédural.

Il est particulièrement utile pour les jeux et autres supports visuels tels que les films. Dans le développement de jeux, le bruit de Perlin peut être utilisé pour toute sorte de matériaux ou de texture ondulante. Par exemple, il pourrait être utilisé pour le terrain procédural (un terrain de type Minecraft peut être créé avec ce bruit, par exemple), les effets de feu, l'eau et les nuages.

Ces effets représentent principalement le bruit de Perlin en 2D et 3D qui sont assez compliqués. Pour notre jeu, une implémentation en 1D suffit, car elle permet par exemple d'avoir un terrain à défilement latéral (comme dans Terraria ou Starbound) ou pour créer l'illusion de lignes manuscrites.

L'implémentation se déroule en trois étapes, il nous faut :

1. Une fonction de bruit qui permet d'associer des valeurs aléatoires à chaque point de l'espace \mathbf{N}^{dim}
2. Une fonction de bruit lissé par interpolation.
3. Une combinaison linéaire de fonctions de bruit lissé.

La fonction `randNoise` représente cette fonction de bruit pseudo-aléatoire, c'est-à-dire une fonction $f : \mathbf{N} \rightarrow \mathbf{N}$ qui, à une valeur donnée, associe une valeur qui semble aléatoire comprise entre -1 et 1.

```
float Perlin::randNoise(int t)
{
    t = (t<<13) ^ t;
    t = (t * (t * t * 15731 + 789221) + 1376312589);
    return 1.0 - (t & 0x7fffffff) / 1073741824.0;
}
```

La deuxième phase de l'algorithme de Perlin consiste en l'interpolation de valeurs intermédiaires définies régulièrement en certains points de l'espace par une fonction de bruit. Concrètement, imaginons que l'on reprenne notre bruit unidimensionnel où l'on a associé à chaque entier une valeur pseudo-aléatoire comprise entre -1 et 1, et que l'on souhaite tracer une courbe continue passant par ces points. On souhaite donc définir une fonction $g : \mathbb{R} \rightarrow \mathbb{R}$ dont la restriction à \mathbf{N} est f . La fonction `interpolate` se compose comme suit :


```
float Perlin::interpolate(float a0, float a1, float w)
{
    return (a1 - a0) * w + a0;
}
```

et effectue une interpolation linéaire entre `a0` et `a1`. `w` représente le poids (weight) et est compris entre 0 et 1.

Nous disposons maintenant de fonctions qui nous permettent d'interpoler entre deux valeurs de notre bruit. Nous pouvons donc écrire une fonction de « bruit lissé ». Elle prend donc en paramètre la coordonnée `x`, dont elle sépare la partie entière et la partie fractionnaire, pour ensuite interpoler entre le point de coordonnée (`x`) et celui de coordonnée (`x+1`).

Ainsi, on boucle la dernière étape avec la fonction `noise` qui crée un bruit de Perlin au temps `t` en utilisant la fonction `randNoise` qui crée un bruit aléatoire depuis `t` ainsi que la fonction `interpolate` qui va appliquer une interpolation linéaire. Finalement, on obtient bien notre bruit de Perlin !

```
float Perlin::noise(float t)
{
    int t0 = (int)t;
    int t1 = t0 + 1;
    float sx = t - (float)t0;

    double n0 = randNoise(t0);
    double n1 = randNoise(t1);

    return interpolate(n0, n1, sx);
}
```

Voici le [site](#) dont nous nous sommes inspirés pour cette méthode que je vous invite à consulter pour plus d'informations !

6.2 Plot de la simulation

L'idée de pouvoir visualiser le résultat de notre simulation fut pour nous très importante car elle constitue tout l'intérêt de notre jeu qui étudie l'évolution de la proportion de loups et de moutons en fonction du temps !

Ainsi, après quelques recherches sur comment faire des plots à partir de vecteurs, on s'est tourné vers la librairie `pbPlots` dont vous pouvez voir le dépôt git. Ainsi, après s'être familiarisé avec la librairie et implémenter les différentes fonctions nécessaires, on obtient le plot suivant dont on est particulièrement fiers !

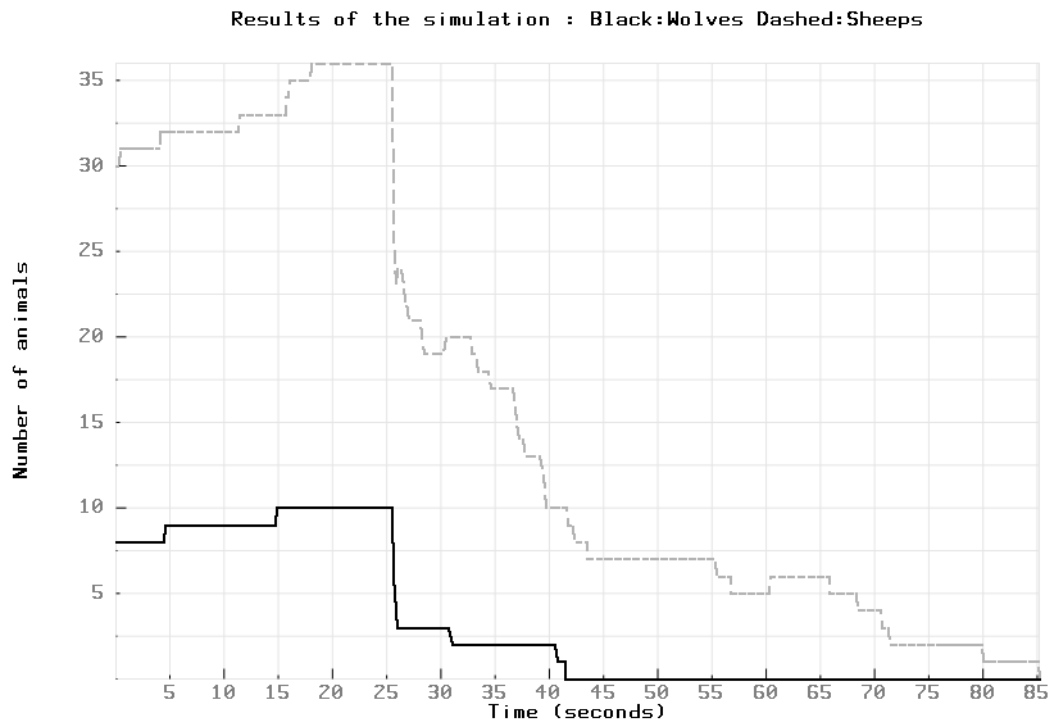


FIGURE 5 – Exemple de plot s’affichant à la fin de la simulation

6.3 La partie audio

Enfin, la dernière partie dont nous sommes satisfaits est la partie audio. En lançant le jeu, nous nous sommes aperçus qu’un jeu avec ou sans son change totalement l’immersion et la satisfaction de l’utilisateur. De plus, SFML comporte pas mal de fonctions prédéfinis et supporte beaucoup de formats audio (ogg,wav,...). Ainsi, nous avons rajouté une classe *Music* comportant toutes les musiques joués dans la fenêtre de lancement et dans la fenêtre de jeu, c’est-à-dire :

- Une musique assez épique au départ avec le hurlement d’un loup en fond.
- Une musique de bataille finale qui se joue en boucle tant que la simulation n’est pas finie.
- Un court son lorsqu’on appuie sur le bouton **START GAME**

Voici la [documentation](#) sur l’implémentation de fichiers audio avec SFML.

7 Conclusion

En conclusion, ce projet fut très enrichissant et agréable à réaliser. Malgré les quelques contraintes du projet, nous avons une assez grande liberté de choix en ce qui concerne notre application. Ainsi, travailler sur sa propre idée ainsi que celle de son partenaire est très satisfaisant car on crée notre futur application en commençant par quelques coups de crayon sur une feuille pour arriver à la fin à un résultat conforme à l'idée de départ !

De plus travailler en équipe nous rappelle que nous ne sommes pas toujours d'accord sur tout et qu'il faut parfois prendre sur soi.

Enfin, ce projet nous a permis de consolider nos connaissances en **C++** ainsi que les nouvelles notions acquises lors des séances de TP et de pouvoir mettre tout sa en oeuvre de manière autonome.

En espérant que ce projet vous aura plu, nous vous remercions pour votre attention !