BIGORRE Noé, CAPMARTIN Léo, DURFORT Alexis, MASSET Aurélien

Groupe Segfault

Rapport de soutenance

Introduction :

Le groupe Segfault est constitué de 4 membres de Strasbourg et Toulouse. A cause de circonstance hors de notre contrôle, nous avons tous dû couper court à notre semestre à l'étranger et revenir en France pour participer au semestre 4 d'EPITA. C'est dans ce contexte qu'a vu le jour le groupe Segfault.
Le but de notre projet est de réaliser un simulateur d'évolution accompagné d'un système d'édition de la simulation pour permettre de recréer une multitude de scénarios avec un unique programme.
Pour cette soutenance, nous avons bien avancé sur ce que nous avions prévu. Cependant, certaines fonctionnalités ne sont pas encore implémentées. Dans ce rapport, nous vous présenterons donc tout ce que nous avons pu faire jusque-là.

Sommaire/

I. Moteur

A. Simulation

Le moteur est l'entité principale qui permet de faire tourner notre simulation. Il fonctionne grâce à la bibliothèque SDL. La manière dont nous l'avons implémenté nous permet d'exécuter une fonction donnée à chaque intervalle de temps. C'est dans cette fonction que toute la logique de la simulation, que nous détaillerons plus tard dans ce rapport, est exécutée. Le moteur exécute donc cette fonction à un intervalle de temps donné. Cet intervalle nous permet de contrôler la vitesse de notre simulation : plus l'intervalle est court, plus la simulation est rapide.
Pour ce faire, nous utilisons la fonction SDL_Delay() qui nous permet de mettre en pause le programme un temps donné en millisecondes.
Pour contrôler la vitesse de la simulation nous avons ajouté un bouton qui possède trois états : lent, médium et rapide. Nous avons aussi ajouté un deuxième bouton qui lui permet de mettre en pause la simulation.
Pour implémenter ces boutons, nous avons une fonction appelée à chaque frame qui s'occupe de gérer tous les événements. Nous pouvons ainsi récupérer facilement le clic de la souris sur l'écran. La position de celle-ci nous permet alors facilement de détecter si l'utilisateur clique sur un bouton ou non, et si oui, lequel.

B. Agents Handler

Pour manipuler les différents agents de la simulation il faut avoir un moyen de les stocker quelque part. Cependant un simple pointeur vers une liste d'agents ne suffira pas car la population évolue beaucoup au fil du temps et utiliser realloc() à chaque fois qu'un agent est ajouté ou enlevé de la liste ne serait vraiment pas optimisé. La solution à ce problème est d'utiliser une liste chaînée pour pouvoir stocker chaque agent de la simulation. L'avantage est qu'avec cette structure l'ajout et la suppression ne requièrent plus de realloc la liste entière, ce qui est un gain énorme en matière d'optimisation, de plus l'implémentation de cette structure est assez simple. Il a donc fallu aussi écrire les fonctions associées à cette structure, c'est-à-dire une fonction initLinkedlist(), push(), pop() et une dernière fonction pour free cette même liste.

II. Agents

A. Principe

Pour rappeler brièvement, les agents sont les entités qui vont peupler notre simulation. Ils ont différentes caractéristiques listées ici :

Caractéristique :	Description :
Espèce	Nom de l'espèce
Temps de vie maximum	Temps de vie (en secondes)
Énergie	Points de vie de l'agent en quelque sorte. L'agent meurt s'ils tombent à 0 (Max : 100).Il pourra se reproduire s'ils dépassent un certain seuil qui sera déterminé. (fixé globalement)
Rayon de détection de nourriture	Rayon dans lequel l'agent détectera sa nourriture
Nourriture(s)	Type de nourriture dont se nourrit l'agent (pouvant elle-même être un autre agent) (fixe)
Vitesse	Vitesse de déplacement de l'agent
Résistance	Valeur entre 0 et 1, déterminant la préférence de température du climat (0 = Froid, 1 = Chaud)
Nombre de la portée maximum	Nombre de descendants maximum lors de la reproduction
Type de reproduction	Sexué (requiert deux agents pour produire un ou plusieurs agents) ou asexué (besoin d'un seul agent qui va se multiplier)
Chance de naissance	Chance que chaque enfant naisse
Coût en énergie de mise-bas	Coût en énergie pour un agent, pour lancer ses tirages pour tenter de donner naissance (fixe)
Coût en énergie de la naissance d'un enfant	Coût en énergie supplémentaire par enfant qui va naître (fixe)

Les trois dernières caractéristiques sont nouvelles. À noter que les deux dernières sont notées fixes car nous ne voulons pas qu'elle puisse évoluer avec les différentes générations.

Pour traduire cela en C nous allons créer plusieurs structures : tout d'abord une structure "*agentType*" ayant toutes les caractéristiques listées précédemment. Voici sa déclaration :

```c
typedef struct agentType{
    char* name;
    float lifeSpan;
    float energy;
    float speed;
    char* reproductionType;
    int maxChildren;
    float fertilityRate;
    float energyBirthCost;
    float energyPerChild;
    float resistance;
    float hearingRange;
    char** targets;
    int targetAmount;
    int dying;
    Uint32 color;
} agentType;
```

Une deuxième structure, avec moins de composants est nécessaire la structure "*Agent*" *:*

```c
typedef struct agent{
    int id;
    agentType* type;
    int Xpos;
    int Ypos;
}agent;
```

Avec ces deux structures, nous sommes capables d'identifier chaque agent, de savoir la position et leurs caractéristiques respectives.

B. Mouvement

Les différents mouvements des agents sont régis par la fonction agentBehaviour qui détermine quel comportement un agent doit adopter en fonction de sa situation.

Son comportement est le suivant : Il va tout premièrement tenter de se reproduire, en cas d'échec, il va par la suite tenter de se nourrir, enfin, si aucune des actions précédentes ne peut être complétée, l'agent va se déplacer vers une position déterminée aléatoirement. Cependant, choisir cette position à chaque appel de fonction donnerait un biais à l'agent qui le forcerait vers le centre, alors, cette position ne sera mise à jour que quand l'agent aura atteint ce point.

C. Reproduction

En programmant la reproduction de deux agents, nous nous sommes rendu compte que les caractéristiques précédentes ne nous convenaient pas. Nous avons alors décidé d'en rajouter 3 : chance de naissance, coût en énergie de mise-bas et le coût en énergie de la naissance d'un enfant. Le principe de la reproduction est donc le suivant : pour chaque enfant qui va potentiellement naître, choisir aléatoirement si la reproduction utilise la dérive génétique ou simplement la reprise des caractéristiques des parents. Si la dérive génétique est choisie, alors pour chaque caractéristique non fixée, nous allons ajouter ou soustraire un certain pourcentage fixé (actuellement 1%) a cette caractéristique. Si la deuxième méthode est choisie alors pour chaque caractéristique non fixée, nous allons choisir aléatoirement si celle-ci prendra la valeur du premier parent ou du deuxième parent.

III. Le Terrain

A. Généralités

Le terrain est un objet qui (en plus de nous fournir un fond d'écran qui n'est pas monochrome) nous permet de définir facilement les zones froides et les zones chaudes. Nous avons donc décidé que les zones les plus sombres du terrain seraient considérées comme des zones chaudes et les zones les plus claires, les zones froides.
Pour générer ce terrain, nous avons choisi d'utiliser le bruit de Perlin.

B. Bruit de Perlin

L'utilisation d'une simple fonction aléatoire, permet la création d'une image complètement aléatoire. Cependant, les pixels n'ont aucune relation entre eux et par conséquent, l'image n'est pas ordonnée.

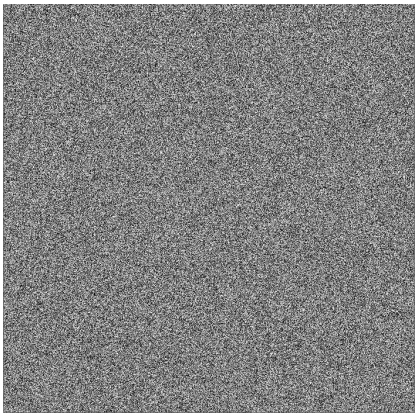
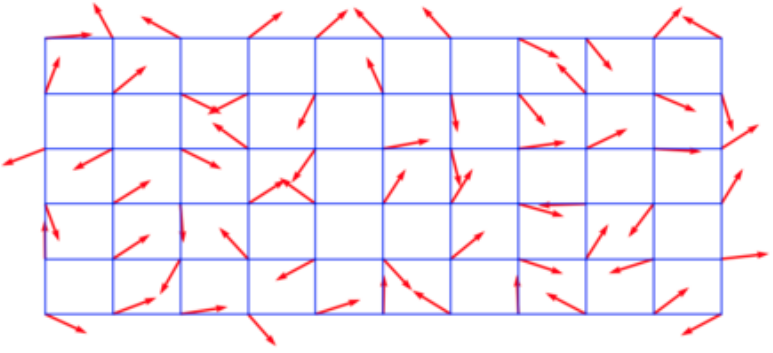


Image générée aléatoirement

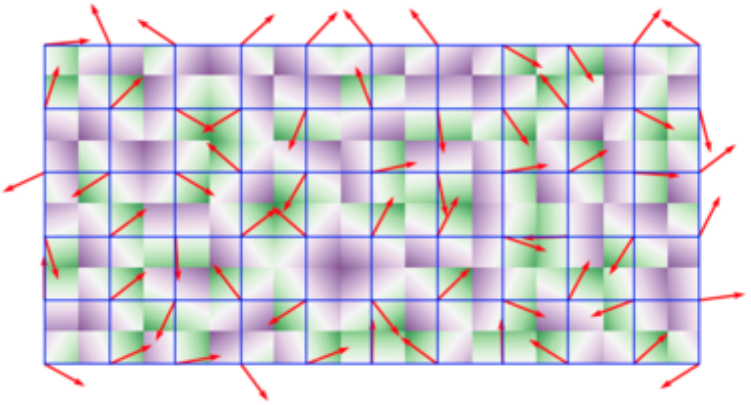
Le bruit de Perlin nous permet d'obtenir une image aléatoire dont les pixels voisins entretiennent une relation de similarité dans leurs couleurs.

Bruit de Perlin généré par notre algorithme

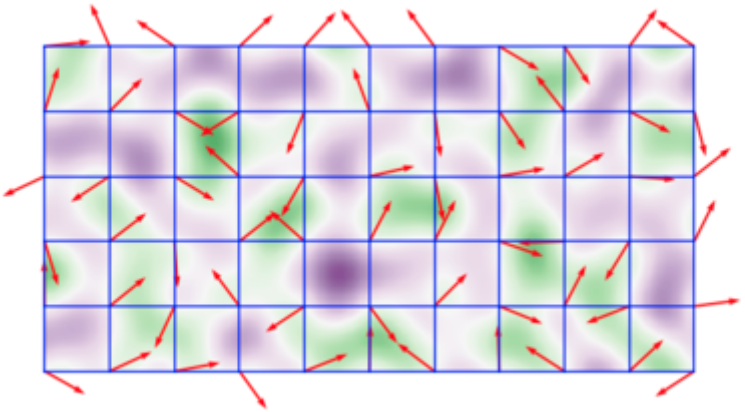
L'image obtenue est alors beaucoup plus uniforme qu'une image aléatoire.
Tout d'abord, on définit une grille de dimension 2 remplie de vecteurs unitaires et de directions aléatoires. Il faut ensuite passer les points de notre image dans une fonction qui va calculer le produit scalaire entre les vecteurs correspondant de la grille et ce point. On récupère ensuite ces résultats que l'on va envoyer dans une fonction d'interpolation. La valeur de retour de l'interpolation est la valeur de perlin pour un pixel donné.



Grille de vecteurs aléatoire



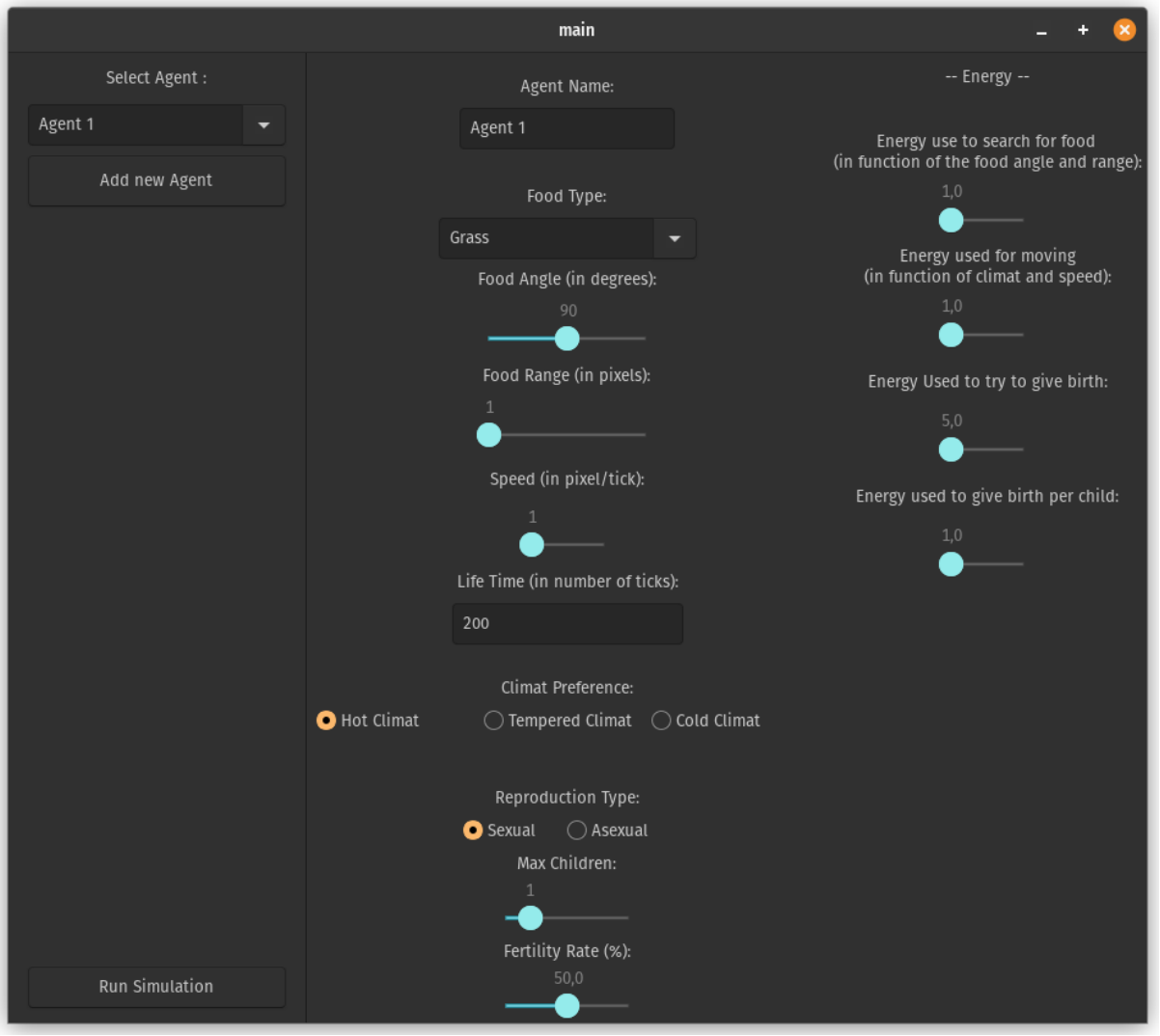
Grille après les produits scalaires (violet = 1, vert = -1)



Grille après interpolation (violet = 1, vert = -1)

IV. Interface

Afin de pouvoir créer et modifier facilement des agents, nous avons commencé, à l'aide de Glade, et de Gtk-3.0, la confection d'une interface utilisateur simple et efficace.



À gauche se situe une liste des unités déjà existantes, ainsi qu'un bouton permettant d'en créer une nouvelle. Enfin, un bouton permettant de lancer la simulation est également présent.

À droite se situent, sur 2 colonnes, les informations d'un agent qui sont modifiables. Ces dernières sont développées dans la partie dédiée aux agents.

Actuellement, seul le bouton pour lancer la simulation a été relié, et lance simplement la simulation de Perlin. À termes, chaque bouton sera relié à une action.

Les menus déroulants des agents seront automatiquement mis à jour lorsqu'un nouvel agent fera son apparition.

Les types de nourriture seront à terme une liste cochable, permettant de sélectionner plusieurs types de nourriture.

Conclusion

L’ambiance au sein du groupe est très bonne et nous avons pu nous organiser sans problème pour nous répartir efficacement les tâches. Cependant, nous n’avons pas pu réaliser tout ce que nous imaginions initialement pour cette soutenance. Il nous manque par exemple toute la gestion des statistiques à récolter pendant la simulation. Il faut aussi finir le développement du comportement de nos agents (la reproduction asexuée n’est pas encore faite par exemple). Malgré ces quelques contretemps, nous sommes quand même satisfaits du travail que nous avons fourni jusque-là.