BIGORRE Noé, CAPMARTIN Léo, DURFORT Alexis, MASSET Aurélien

Groupe Segfault

Rapport de Projet

Introduction

Le but de ce 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.

Les simulations d'évolutions sont relativement nombreuses et visent des domaines différents de l'évolution. On y retrouve les modèles usuels comme ceux de dynamique de population, de dérive génétique ou encore de sélection naturelle et artificielle. À ces modèles viennent s'ajouter d'autres moins communs comme l'héritage de paterne ou encore l'automate cellulaire. On remarque cependant que parmi toutes ces simulations le modèle est fixe et ne peut pas être modifié, mis à part ces paramètres.

Ainsi, en l'état, créer un nouveau modèle de simulation requiert soit de partir de zéro, soit de se baser sur un ancien modèle et modifier son code.

La grande majorité de ces simulations utilisent des algorithmes évolutionnistes reposant sur la programmation génétique. Comme son nom l'indique, elle se base sur le principe de sélection naturelle pour permettre aux agents d'évoluer et de s'adapter à la situation. La principale force de cette technique est sa proximité avec la réalité, ce qui permet de rendre la simulation plus réaliste. Cet avantage vient avec l'inconvénient d'être plus coûteux en temps de calcul.

Sommaire/

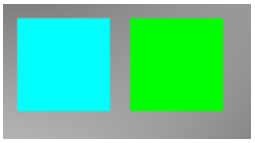
I. Moteur

A. Simulation

Le moteur à été développé par Léo Capmartin. Il 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. Modifications des boutons

Lors de la précédente soutenance, notre moteur nous permettait de mettre la simulation en pause, d'accélérer la vitesse ou de la ralentir. Cependant les boutons permettant d'effectuer ces actions étaient monochromes, ce qui n'aide évidemment pas à la compréhension du fonctionnement de notre application.



Premières versions des boutons

Pour remédier à ce problème nous avons décidé d'utiliser des images permettant de savoir rapidement quel est le rôle d'un bouton.

Différentes versions des nouveaux boutons

Lors de l'implémentation nous nous sommes heurté à un problème majeur. Lorsque nous mettions en pause la simulation, lors du clic sur un bouton, la nouvelle image de celui-ci venait se superposer sur l'ancienne image au lieu de la remplacer. Il est important de noter que ce problème intervient uniquement lorsque la simulation est en pause.

Exemple de boutons qui se superposent

Le problème vient du fait que nous dessinons toutes les frames sur la même image que celle qui est rendue à l'écran. Lorsque la simulation est en marche, la nouvelle image vient recouvrir celle des boutons et nous affichons ensuite les boutons par-dessus. Cependant dès que

nous mettons en pause la simulation, nous ne mettons pas à jour la simulation et donc nous ne redessinons pas de nouvelle image, par conséquent au moment de dessiner les boutons, ils viennent s'afficher au-dessus des précédents.

Pour corriger le problème, nous utilisons donc deux images : une pour dessiner toutes les mises à jour chaque frame et une qui sera affichée toutes les frames. Pour ce faire, l'image des mises à jour est dessinée sur celle qui est affichée puis nous dessinons les boutons sur l'image à afficher. Le fait de réaliser ces deux étapes quoi qu'il arrive (que la simulation soit en pause ou non) nous permet d'éviter ce bug.

Nous avons aussi ajouté un bouton "stop" permettant, comme son nom l'indique, d'arrêter la simulation.

Bouton "stop"

c. Agents Handler

Le Agent hendler a été développé par Alexis Durfort.

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

Les caractéristiques des agents ont été réfléchies et améliorées par tout le groupe. Les agents sont les entités qui vont peupler notre simulation. Ils ont différentes caractéristiques listées ici :

Caractéristique :	Description :
Espèce	Nom de l'espèce (fixe)
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 au climat	Valeur entre 0 et 1, déterminant la préférence de température du climat (0 = Froid, 1 = Chaud)
Nombre d'enfant 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) (fixe)
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

Cette partie a été majoritairement développée par Noé Bigorre avec quelque ajouts de Alexis Durfort

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.

Pendant la deuxième soutenance nous avons modifié le comportement de la fonction qui permet de déplacer l'agent vers un point aléatoire, L'inconvénient de cette technique est que l'agent avait souvent besoin de traverser une grande partie du terrain avant d'atteindre son point ce qui, de notre point de vue, n'était pas très naturel. Régler ce problème était heureusement assez simple, au lieu de prendre un point aléatoirement sur le terrain, on va prendre un point aléatoirement autour de l'agent pour qu'il puisse changer de point plus souvent et éviter de parcourir tout le terrain inutilement. Cependant un inconvénient de cette technique est que les agents auront plus tendance à rester autour de leur position de départ.

C. Reproduction

Cette partie a été développée par Alexis Durfort.

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.

Nous nous étions arrêtés là pour la première soutenance et n'avions pas fait la deuxième façon qui est la reproduction asexuée des agents. Nous l'avons alors implémenté pour la deuxième soutenance, cela n'a pas été aussi difficile que pour la reproduction sexuée étant donné que nous avons principalement utilisé des fonctions déjà écrites. Le principe est le même que celui d'avant, sauf que nous n'avons pas deux différents agents donc nous ne pouvons qu'effectuer la dérive génétique sur la descendance de l'agent ce qui facilite grandement les choses.

D. Maladies

Cette partie a été développée par Noé Bigorre.

L'implémentation des maladies dans les agents est faite par le biais de la *struct sickness* qui permet de contrôler les différentes maladies et les interactions entre elles.

Cette structure comporte les éléments suivants :

Le nom de la maladie	
l'infectiosité	Représente la chance qu'une maladie se propage à un agent à proximité.
La sévérité	représente le coût additionnel en énergie qu'un agent doit fournir à cause de la maladie.
La mortalité	représente la chance qu'un agent a de mourir après une action.
canInfect	la liste des agents que la maladie peut infecter.

Les différentes maladies des agents sont représentées dans une liste chainée se comportant comme un ensemble pour éviter les duplications de maladies sur un même agent.

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'implémentation du bruit de Perlin à été réalisée par Léo Capmartin.

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.

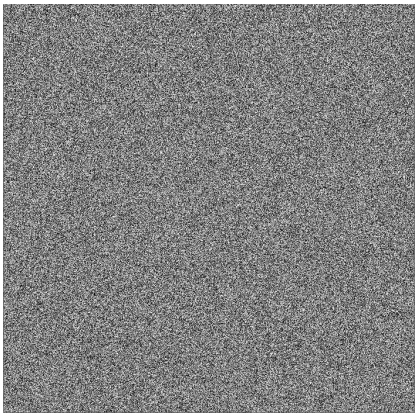


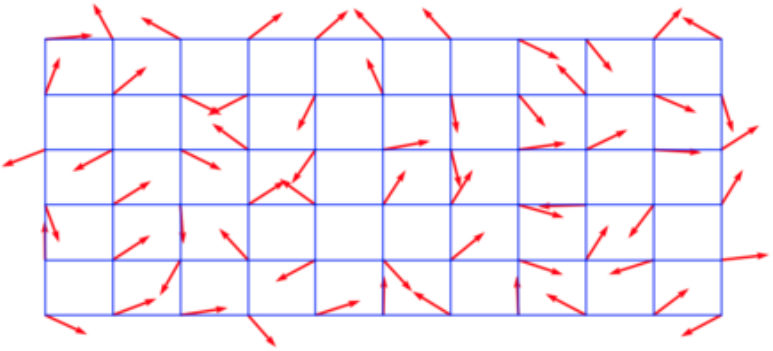
Image générée aléatoirement

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

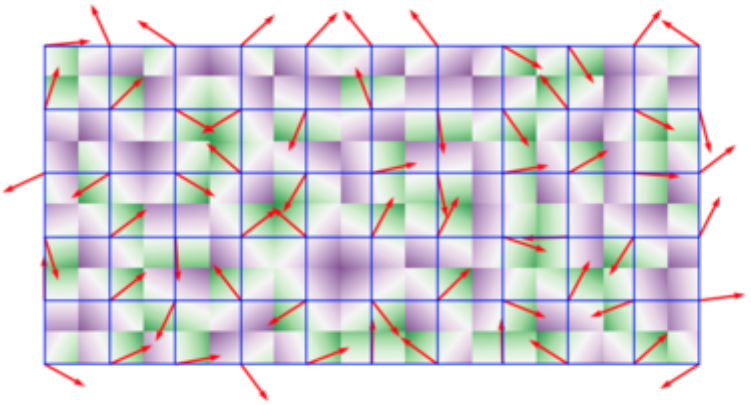
Bruit de Perlin généré par notre algorithme

L'image obtenue est alors beaucoup plus uniforme qu'une image aléatoire.

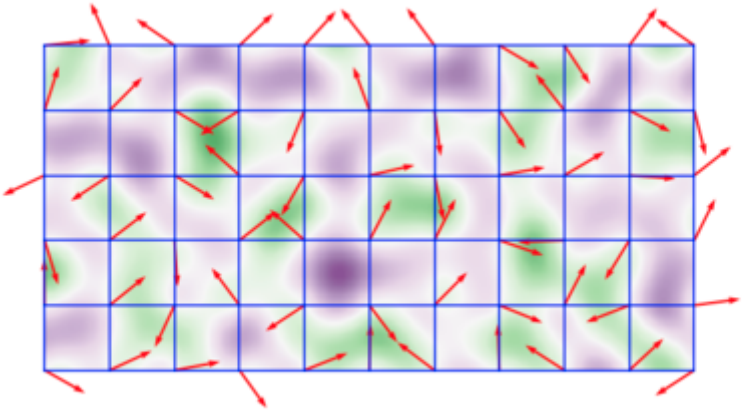
Tout d'abord, on définit une grille de dimension 2 remplie de vecteurs unitaires et de directions aléatoires. Il faut ensuite passer les points de notre image dans une fonction qui va calculer le produit scalaire entre les vecteurs correspondant de la grille et ce point. On récupère ensuite ces résultats 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)

c. Génération de la nourriture

Cette partie a été développée par Alexis Durfort.

Pour la première soutenance, nous nous étions arrêtés à l'ajout d'un système de bruit de perlin pour créer un terrain ou nos agents peuvent évoluer. Pour cette soutenance nous avons donc ajouté la génération de la nourriture au terrain. L'idée est la suivante, nous allons en premier temps générer des arbres aléatoirement sur le terrain, puis à chaque tick de la simulation, ces arbres vont générer des fruits qui pourront être mangés par les agents herbivores. Pour l'implémentation nous avons utilisé plusieurs stucture: d'abord la structure pour les arbre qui possède 4 caractéristique :

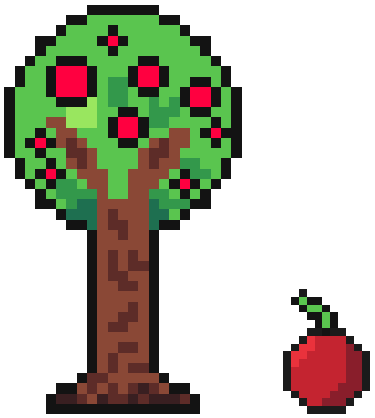
Xpos	Position en X
Ypos	Position en Y
radius	Rayon d'apparition des fruit
currentFood	Nombre de fruit actuellement autour de l'arbre

Nous avons ensuite une deuxième structure pour les fruits qui possède comme caractéristique:

Xpos	Position en X
Ypos	Position en Y
energyToGive	Quantité d'énergie a donné à l'agent si il mange ce fruit
lifeSpan	Durée de vie du fruit

Nous avons une dernière structure qui n'est ni plus ni moins qu'une liste pour accéder au différent fruit actuellement sur le terrain.

Pour dessiner les arbres et les fruit nous avons aussi créer des sprite nous même:



D. Interaction entre les agents et le terrain

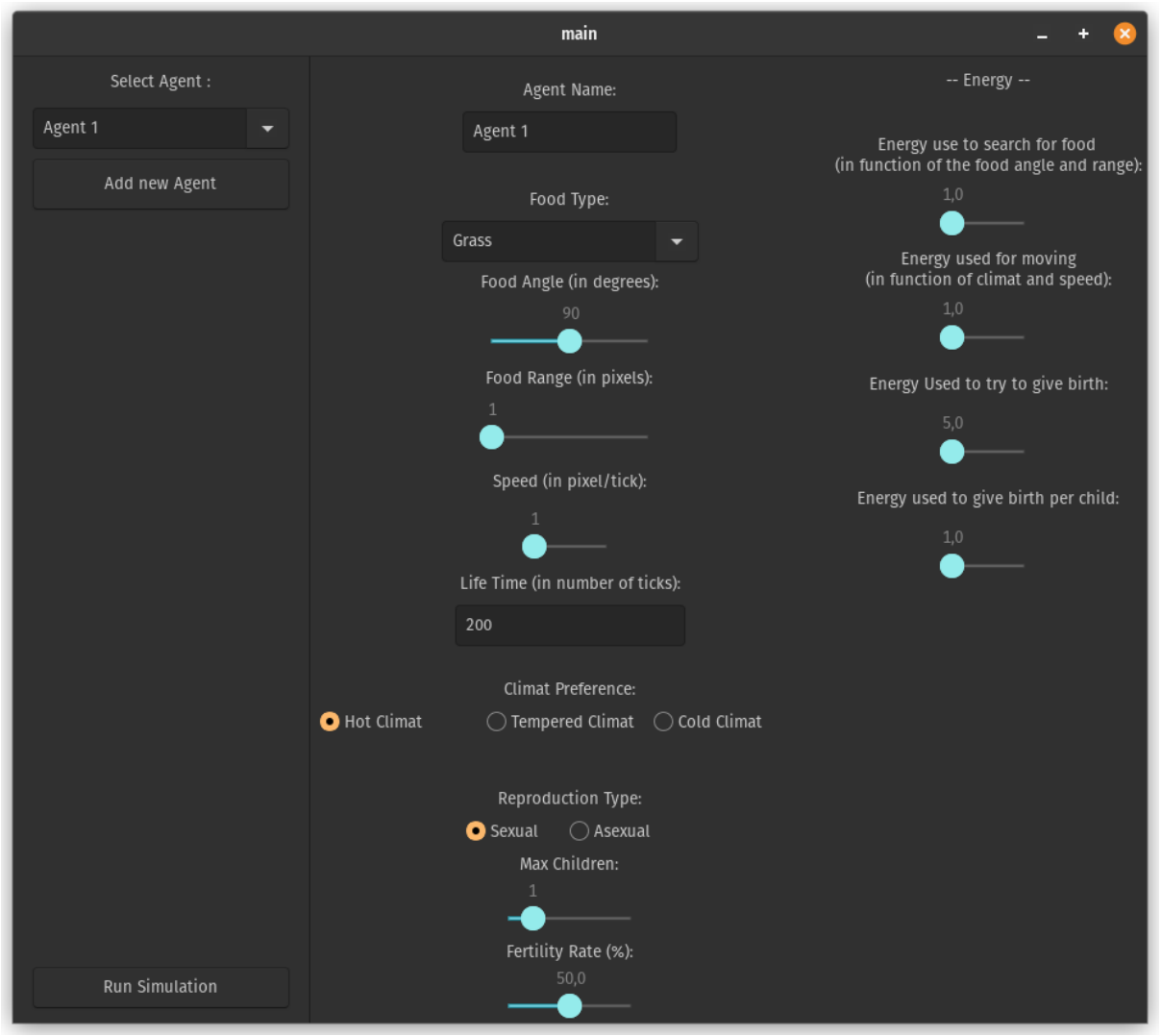
Cette partie a été développée par Noé Bigorre.

Le lien entre le terrain et l'agent a été ajouté au niveau du climat, ce qui permet d’augmenter le coût en énergie des actions quand l’agent se trouve dans un climat trop différent de son climat préféré.

Plus la différence entre le climat et celui préféré de l’agent est important, plus le coût additionnel en énergie est important, car on calcule le carré de la différence entre les deux. Ainsi, pour de petites différences, le changement du coût est relativement faible, mais pour de grandes différences, le coût peut sérieusement grimper.

IV. Interface

L'interface a été réalisée par Aurélien Masset.



Interface lors de la première soutenance

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

À gauche se situe une liste des unités, plantes et maladies 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, d'une plante ou d'une maladie qui sont modifiables. Ces dernières sont développées dans leurs parties respectives, développées précédemment.

Les boutons sont bels et bien reliés, lançant des signaux permettant de faire l'action demandée. Lors de la création, un bouton final a été rajouté, afin que l'on ne récupère les informations choisies qu'à ce moment-là pour envoyer le tout vers un constructeur.

L'interface reliant le signal du bouton pour lancer la simulation à une fonction.

```
void call_run(GtkButton *button, gpointer user_data)
{
```

```
GtkButton* run = GTK_BUTTON(gtk_builder_get_object(builder, "Run"));
g_signal_connect(run, "clicked", G_CALLBACK(call_run), NULL);
```

Extraits des Signal handler de notre interface

```
<object class="GtkButton" id="Run">
  <property name="label" translatable="yes">Run Simulation</property>
  <property name="visible">True</property>
  <property name="can-focus">True</property>
  <property name="receives-default">True</property>
  <property name="valign">start</property>
  <property name="margin-start">4</property>
  <property name="margin-end">4</property>
  <property name="margin-bottom">2</property>
</object>
```

Extrait du code de notre interface, correspondant au signal traité au-dessus

Les menus déroulants se mettent à jour en fonction des nouveaux éléments créés, ainsi que de leur type *ItemType*, nouveau paramètre ajouté, propre au comportement de l'interface, afin qu'elle puisse séparer les agents des plantes et des maladies.

ItemType est un paramètre choisissable lors de la création d'un nouvel agent. Il n'est plus modifiable après. Ainsi, il est impossible de transformer une plante en maladie.

V. Statistiques

A. Généralité

L'implémentation des statistiques a été faite par Léo Capmartin. Notre projet donne aux statistiques une grande importance : en effet, sans cette partie, notre projet n'aurait que peu d'intérêt.
À chaque intervalle de temps de x secondes, nous enregistrons l'état de la simulation ce qui permet par la suite de pouvoir constater les évolutions de nos espèces. Pour ce faire, nous avons une liste chainée dans laquelle, à chaque intervalle de temps, nous enregistrons une copie de l'état de la simulation. Il nous suffit alors de parcourir cette liste pour avoir une idée de l'évolution de la simulation.

B. Enregistrement des données

Une fois la simulation terminée, nous parcourons la liste pour enregistrer proprement toutes les données dans un fichier CSV qui pourra ensuite être lu par un logiciel comme excel (ou équivalent) pour effectuer tous les traitements que l'utilisateur souhaite.

```
time (secondes),1,2,3,4,5,6,7

espece 1
gene 1,56,64,12,0,-22,-44,-66
gene 2,56,84,23,21.333333333333329,4.8333333333333286,-11.666666666666671,-28.166666666666671
...

espece 1
gene 1,56,64,12,0,-22,-44,-66
gene 2,56,84,23,21.333333333333329,4.8333333333333286,-11.666666666666671,-28.166666666666671
...
```

Données après le traitement dans un document CSV

	A	B	C	D	E	F	G	H	I
1	time (secondes)	1	2	3	4	5	6	7	
2									
3	espece 1								
4	gene 1	56	64	12	0	-22	-44	-66	
5	gene 2	56	84	23	21,33333	4,833333	-11,66667	-28,16667	
6	...								
7									
8	espece 1								
9	gene 1	56	64	12	0	-22	-44	-66	
10	gene 2	56	84	23	21,33333	4,833333	-11,66667	-28,16667	
11	...								

Données affichées dans un tableau excel

VI. Site web

Le site web a été développé par Léo Capmartin. Pour notre site, nous avons décidé de réaliser un design s'inspirant d'un terminal Linux.
https://leocapmartin.github.io/segfaultsite.github.io/

Plusieurs commandes sont disponibles permettant de naviguer sur le site. En voici une liste non exhaustive :

- help : permet d'afficher les commandes importantes ainsi que leur description.
- about : permet d'afficher une courte description du projet ainsi que de notre groupe
- clear : permet d'effacer tout ce qui est affiché sur le terminal
- files : permet d'afficher plusieurs liens permettant de télécharger nos rapports ou notre projet
- github : redirige vers notre page github et affiche le lien du github si l'utilisateur à un bloqueur de pop-up
- logo : affiche le logo de notre groupe
- welcome : affiche le message de bienvenue de notre site

Pour développer notre site nous avons utilisé du HTML, CSS et JavaScript très simple.

VII. Répartitions des tâches

Tâche	BIGORRE	CAPMARTIN	DURFORT	MASSET
Interface				x
moteur		x		
simulation	x		x	
terrain :		x	x	
lien entre les tâches :				x
Site Web		x		

VIII. Points de vue personnels

- *Bigorre Noé :*

Participer à ce projet a été pour moi une expérience enrichissante et très amusante notamment au niveau de la conception et du brainstorming. Le travail sur la gestion des agents aura été passionnant et voir les agents se comporter correctement aura été vraiment fantastique et les bugs étaient eux particulièrement amusants à regarder. Apprendre à approcher une méthode proche de l'orienté objet dans un langage tel que le C aura également été très instructif.

- *Capmartin Léo :*

Ce projet était très instructif à imaginer, et à brainstormer. Il a été pour moi très instructif. Il m'a permis de comprendre ce qu'est le bruit de Perlin et de comment l'implémenter, ce qui est un quelque chose que j'avais déjà envie de faire avant le début du projet. J'ai aussi appris à optimiser l'organisation de données. J'ai aussi pu apprendre les

fondamentaux du fonctionnement d'un moteur graphique. Même si celui que nous avons implémenté est très rudimentaire, il m'à donné envie de creuser plus en profondeur ce sujet.
De manière générale, je suis très satisfait de ce projet qui m'a permis d'acquérir de nouvelles connaissances.

- ***Durfort Alexis :***

J'ai beaucoup aimé imaginer le projet, brainstormer sur ce que notre simulation pourrait faire etc.. De plus, travailler sur ce projet m'a vraiment permis d'approfondir ma connaissance du C et le fonctionnement des structures. Nous avions déjà travaillé en C pour l'OCR mais ce projet m'a vraiment permis de me rendre compte à quel point l'architecture du code est importante, surtout pour la lisibilité du code. De plus ce projet m'aura aussi permis d'avoir une petite idée de ce que travailler en équipe et en distanciel signifie, car dans les projets précédents nous étions toujours avec des gens que nous connaissions ou, au minimum travaillaient au même endroit que nous, alors qu'ici nous avons des gens de plusieurs régions différentes et que personnellement je n'avais jamais rencontrés (à part Noé bien sûr).

- ***Masset Aurélien :***

Le projet était intéressant à imaginer, et à brainstormer. il avait plus de profondeur que ce que l'on pourrait penser. Ne serait-ce qu'imaginer comment nos agents pourraient vivre, mourir, et surtout *survivre*, nous a forcé à imaginer un système d'énergie, d'espérance de vie, et pléthore d'autres statistiques.

Bien que j'aie particulièrement travaillé sur l'interface, ainsi que les liens avec les parties des autres, j'ai tout de même participé au même titre que les autres à la conception et l'imagination de tout ceci.

Cependant, j'aurais aimé pouvoir aider un peu plus mes camarades. J'ai eu divers problèmes techniques durant les quelques derniers mois (ma WSL qui a rendu l'âme, une impossibilité d'utiliser une VM suite à un problème encore et toujours inconnu en rapport avec la

virtualisation que je n’ai toujours pas été capable de régler). Ces problèmes ont rendu ma participation vraiment difficile, voire problématique.

Cependant, mes camarades ont su trouver des moyens d’outrepasser ces problèmes (me faire travailler sur l’ordinateur de l’un d’eux par exemple). Je suis malgré tout très content du projet que nous avons pu produire, ainsi que du groupe que nous avons formé.

Très bonne ambiance, et bonne productivité.

Conclusion

De manière générale, nous avons beaucoup apprécié ce projet. Il nous à donné envie de continuer à développer notre simulation et de la complexifier encore plus. Nous aimerions par exemple pouvoir ajouter des paramètres globaux venant modifier complètement le fonctionnement de la simulation. Malgré les difficultés rencontrées (bugs en tous genres, segfaults ou encore difficulté matérielle), nous sommes tout de même très satisfaits du projet que nous avons réussi à créer.