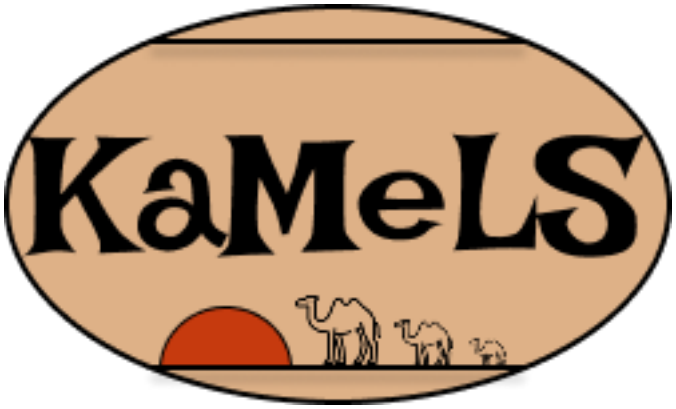
Rapport de Soutenance Finale - Projet S4

Hayvolution

Groupe : KaMeLS



Kylian Djermoune Maxime Trimboli Lino Joninon Simon Campredon

Table des matières

1 Introduction **4**
1.1 Présentation du projet 4
1.1.1 Inspirations 5
1.1.2 Aspect technique 8
1.2 Présentation du groupe 9
1.2.1 Présentation des membres 9
1.3 Objectifs du projet 11
1.3.1 Environnement cohérent 11
1.3.2 Équilibrage de la simulation 11
1.3.3 Observer l'évolution des espèces 11
1.3.4 Organiser le comportement des animaux 12
1.3.5 Application ergonomique et personnalisable 12

2 Développement du projet **13**
2.1 Environnement 13
2.1.1 Mise en place 13
2.1.2 Génération 14
2.1.3 Biomes 15
2.2 Animaux 18
2.2.1 Attribut des animaux 18
2.2.2 Mutations et évolution 19
2.2.3 Généalogie 20
2.2.4 Interactions 21
2.2.5 Déplacement 24
2.3 Gestion de la simulation 26
2.3.1 Pendant la simulation 26
2.3.2 Fin d'une génération 27
2.3.3 Équilibrage 27
2.4 Statistiques 29

2.4.1 Nombre d'individus et caractéristiques moyennes 30
2.4.2 Affichage des statistiques 32
2.5 Interface 33
2.5.1 Interface et boutons 33
2.5.2 Fenêtre graphes 34
2.5.3 Ajout d'espèces 35
2.6 Site web 35

3 Conclusion **38**
3.1 Avancement atteint 38
3.1.1 Comparaison avec notre idée initiale 38
3.1.2 Rajouts et abandons d'idées 39
3.2 Difficultés rencontrés 40
3.3 Améliorations possibles 41
3.4 Conclusion 42

1 Introduction

1.1 Présentation du projet

Hayvolution (mot-valise entre *hayvon* signifiant *animal* en ouzbek et *évolution*) est un projet de simulation d'évolution d'écosystème qui possède ses propres règles, un jeu de la vie beaucoup plus poussé. Au sein de cet environnement représenté en caractères ASCII, différentes espèces animales évoluent. L'idée de ce projet nous est venue des vidéos de la chaîne YouTube *Primer*, que nous détaillons plus bas. Les animaux ayant tous des caractéristiques différentes, pouvoir observer leurs évolutions au fil des générations et essayer de les prédire a un intérêt certain pour l'utilisateur. L'objectif de ce projet est aussi d'essayer de reproduire de manière simpliste l'évolution ainsi que ses mécanismes, et ce peu importe les choix de départ de l'utilisateur, il s'agit donc de créer un système cohérent et équilibré. Dans ce projet se mélangent des notions de génération aléatoire, pour générer un environnement et pour provoquer des mutations au sein des espèces, mais un des objectifs du projet est de simuler l'intelligence des animaux, notamment grâce à algorithme d'heuristie. Enfin, Hayvolution possède également une interface graphique pour observer le déroulé de la simulation lancée, ainsi qu'une interface dédiée à des statistiques sur la simulation pour observer globalement l'évolution des espèces en fonction de leur population, génération par génération.

1.1.1 Inspirations

Jeu de la vie

Le jeu de la vie imaginé par John Horton Conway en 1970 est un jeu de simulation mathématique que nous connaissions puisque nous l'avons nous même codé l'année dernière. Il nous a permis de nous rendre compte que des règles basiques d'interactions entre les éléments qui sont appelées "cellules" dans le jeu de la vie peuvent mener à une grande complexité. Le jeu de la vie tente à sa façon, comme nous, de répliquer des phénomènes biologiques réels à l'aide de règles mathématiques. Notre affichage simple sous forme de matrice dont on affiche uniquement les caractères nous vient aussi de là.

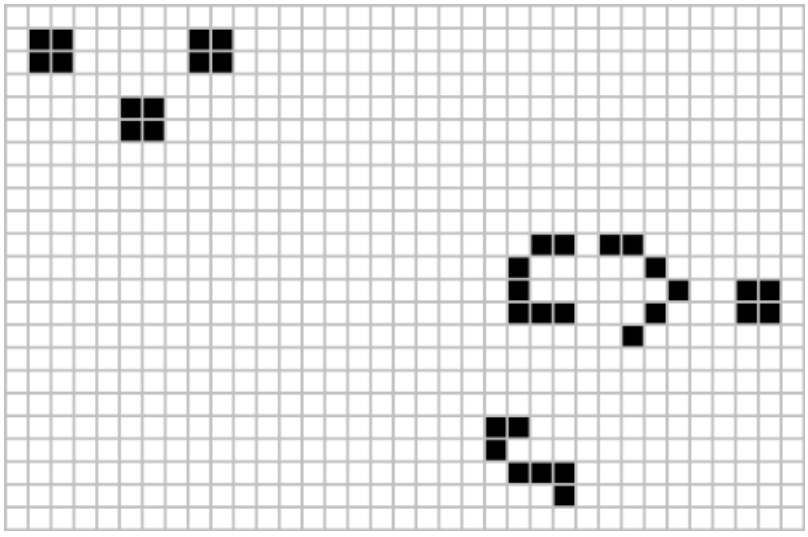


Figure 1 – Le jeu de la vie de John Horton Conway

Primer

Primer est le nom d'une chaîne sur youtube qui propose des contenus de vulgarisation scientifique sur l'évolution nottament. Ces vidéos sont illustrés grâce à des simulations et des petits personnage en 3D que l'on retrouve dans toutes vidéos. Dans sa vidéo "Simulating natural selection", Primer crée un système d'environnement avec une quantité de nourriture limitée. Dans cette environnement évolue une espèce où les individus sont mis en compétition entre eux pour voir si certains individus sont avantagés par rapport à d'autres.

Figure 2 – Vidéo sur la sélection naturelle

Spore

Spore est un jeu de simulation d'évolution d'un être vivant réalisé par le créateur des Sims sorti en 2008. L'objectif de ce jeu est d'amener une espèce à évoluer pour fonder une civilisation, passant d'une simple cellule, à un animal, jusqu'à un empire s'étendant sur toute une galaxie. Le principe

d'évolution a inspiré notre idée de mutation s'effectuant sur chaque génération. Spore pousse le principe de l'évolution assez loin en permettant au joueur après plusieurs épreuves de faire grandir une partie du corps de notre créature ou lui rajouter des attributs. Il doit ainsi décider entre vivre une vie pacifique en étant herbivore capable de survivre aux attaques d'autres créatures, ou bien se concentrer sur la force pour pouvoir manger le plus possible en combattant ses ennemis. Spore nous a donné l'idée de mettre en place plusieurs espèces et la fonctionnalité proie/prédateur.

Figure 3 – Vue de la cellule en évolution dans le jeu Spore

1.1.2 Aspect technique

Ce projet est entièrement réalisé en langage C comme exigé. Les parties de gestion des animaux et de l'environnement peuvent paraitre simple mais utilisent des algorithmes relativement complexes, pour la génération des biomes qui va aussi faire appel à des fonctions mathématiques pour le bruit de perlin, et pour les animaux l'algorithme de décision.
Nous avons du créer aussi beaucoup de structures pour représenter les éléments du projet : la struct Animal, la struct Environment. Mais les structs nous servent aussi d'outils comme la liste qui nous a permis d'implémenter une queue et nous permet de mettre en place la représentation par arbre généalogique. Enfin la struct Stats qui sert à regrouper des statistiques sur la simulation en cours et la struct Sim qui permet de regrouper tous les éléments nécéssaires à la simulation.
Pour ce qui est de l'interface, nous avons évidemment utilisés GTK ainsi que Glade pour nous faciliter la mise en place des différents widgets. Pour la fenêtre des graphes nous avons décidé d'utiliser la librairie GLG que nous avons trouvé sur GitHub en cherchant des façons d'afficher des graphiques sur gtk.

1.2 Présentation du groupe

Nous avons décidé de nommer notre groupe KaMeLS pour Kylian, Maxime, Lino et Simon ainsi qu'en référence au langage de programation OCaml. Nous avons formé ce groupe par défaut et aucun de nous n'avait travaillé avec un autre membre du groupe auparavant. Par chance, nous nous sommes bien trouvé et chaque membre du groupe a su apporter ses connaissances et ses compétences pour réaliser ce projet.

1.2.1 Présentation des membres

Simon Campredon

Pour le projet OCR de S3, je me suis occupé du réseau de neurones, cela a été difficile mais j'ai tout de même de bons souvenirs car il m'a permis de me découvrir une nouvelle passion pour l'intelligence artificielle. C'est pourquoi ce projet est pour moi important car il me permettra possiblement d'élargir mes connaissances déjà acquises ou bien d'en découvrir de nouvelles. Je m'occuppe donc surtout des animaux et de leurs comportements.

Kylian Djermoune

Je suis impatient de travailler sur ce projet de S4, l'idée d'un simulateur d'écosystème m'a directement emballé. La partie dont je m'occupe, l'environnement et notamment sa génération procédurale m'apporteront des compétences nouvelles en programmation. Je me suis occupé de la création de l'environnement et de la génération des biomes.

Lino Joninon

Lors de projets que nous avons effectués précédemment, j'ai senti un manque d'organisation dans les groupes dont je faisais partie. Nous obtenions un résultat fini, mais qui aurait pu être fait plus facilement et plus rapidement avec plus de communication et d'organisation. Donc, en plus de développer mes compétences de codage, je souhaite approfondir et affiner mes capacités à travailler et à interagir au sein d'une équipe. Ma contribution pour le projet est l'interface et lier les différentes parties avec l'interface. Comparer au projet OCR une difficulté supplémentaire dans l'interface est l'affichage constant de l'environnement.

Maxime Trimboli

Nous allons de nouveau réaliser un projet libre, et après le projet S2 et l'expérience que nous avons acquise en programmation, ce projet m'a permis de tester librement beaucoup de choses en C. Un des enjeux de ce projet est de pouvoir reproduire des concepts évolutifs réels, et j'ai beaucoup aimé avoir l'impression de faire un cours de SVT du lycée tout en programmant pour ce projet. Je me suis occupé en partie des animaux, ainsi que des statistiques et de l'interface de la simulation.

1.3 Objectifs du projet

Lorsque nous avions décidé de la teneur de notre projet au début de ce semestre, nous avions diverses inspirations et objectifs précis en tête.

1.3.1 Environnement cohérent

Pour l'environnement, nous souhaitions obtenir un environnement avec différents biomes, où les animaux peuvent se mouvoir, ainsi que la présence d'arbres pour leur permettre de se nourrir.

1.3.2 Équilibrage de la simulation

Pour chacune des espèces animales présente de base dans la simulation, nous voulions configurer des statistiques équilibrées afin de s'assurer que chacune des simulations ne se déroule pas d'une manière similaire. Nous voulions qu'une imprévisibilité soit garantie et que chaque espèce puisse, dans une configuration de base générée aléatoirement, évoluer différemment.

1.3.3 Observer l'évolution des espèces

Nous avons voulu créer un système de mutations simplistes pour essayer de répliquer la sélection naturelle. De manière aléatoire, un nouveau né va recevoir les statistiques de son parent avec de légères variations. L'objectif est que si pour une espèce une l'amélioration d'une certaine caractéristiques donne un avantage de survie, on va retrouver globalement cette tendance pour tous les membres de cette espèce.

1.3.4 Organiser le comportement des animaux

Pour le comportement, nous voulions faire en sorte que les animaux puissent faire leurs choix de déplacement en fonction de leur portée de vision. Qu'ils choisissent l'option la plus avantageuse pour eux en fonction de leur niveau de satiété, des distances, etc...

1.3.5 Application ergonomique et personnalisable

Nous voulions faire une interface pratique et qui permet de personnaliser l'environnement et les espèces présentes dans la simulation. De plus il fallait absolument faire la fenêtre statistiques car elle est primordiale pour donner un aperçu global de ce qu'il se passe dans la simulation.

2 Développement du projet

2.1 Environnement

2.1.1 Mise en place

L'environnement est un des piliers du projet Hayvolution, celui-ci permet de contenir l'ensemble des animaux et autres éléments de la simulation. En premier lieu, nous avions imaginé l'environnement comme pouvant être simplement implémenté à l'aide d'une matrice de caractères. Nous utilisions alors une struct possédant trois attributs ; une matrice de caractères dont la longueur et la largeur sont également des attributs de la structure. Cependant, nous avons réalisé que la matrice de caractère n'était finalement pas viable pour lier les animaux et l'environnement. En effet, l'environnement devait à terme pouvoir contenir des caractères représentant non seulement les animaux, mais également des caractères représentant le biome d'un endroit précis ainsi que les arbres nécessaires à l'alimentation des herbivores. De ce fait, nous l'avons donc remplacée par une matrice de struct Case. Ces struct Case ont, elles aussi, trois attributs ; un pointeur vers un animal, un caractère représentant le décor (la présence d'un arbre ou non) et un caractère représentant le biome. Ainsi, sur une case de coordonnées (x, y) peuvent se retrouver trois éléments. Cela a amené à la modification de la fonction « print_environment » qui peut être appelée avec un paramètre influant sur la couche à afficher.
Après cela, nous nous sommes également rendus compte que la création d'une classe Tree était nécessaire afin notamment d'utiliser des fonctions

sur les arbres telles que « eat_tree » ou « refill_tree ». En effet, ces fonctions utilisent l'attribut counter implémenté dans la struct Tree qui permet de rendre les arbres à nouveau comestible au bout d'un certain nombre de tours de génération. L'attribut consumable nous a permis de faire en sorte que les herbivores ne soient pas attirés vers des arbres qui ont déjà été mangés.

2.1.2 Génération

En ce qui concerne la génération de l'environnement, nous utilisions au départ une simple fonction qui plaçait sur chaque case un arbre avec un certain pourcentage de chance, ce pourcentage se voyant augmenté si des arbres étaient déjà aux alentours de la case en question. Cependant, cette génération des arbres a changé après l'ajout de la fonction de bruit qui sera expliquée dans la partie suivante. Pour ce qui est de l'apparition des espèces animales au début de la simulation, celles-ci apparaissent lors de l'appel à la fonction « spawn_animals ». Cette fonction choisit un emplacement aléatoire au sein de la matrice de l'environnement pour faire apparaître un individu de chaque espèce puis le nombre d'individus souhaité par l'utilisateur autour de ce premier animal. Cependant, nous avons quelque peu modifié le fonctionnement de cette fonction afin de garantir des meilleurs résultats dévolution, en effet, lorsque chaque espèce apparaît à un certain endroit aléatoire de l'environnement, il arrivait parfois que les prédateurs apparaissent trop loin des herbivores entraînant leur extinction de façon prématurée. Ainsi, nous avons décidé de créer un système de foyers, qui permet de faire apparaître au début chaque espèce par groupe de cinq. Par exemple, si l'utilisateur souhaite commencer avec 8 renards, alors 5 d'entre

eux apparaîtront aux alentours d'un certain emplacement, et 3 autres à un autre endroit.

2.1.3 Biomes

La génération cohérente des biomes est possible à l'aide d'une fonction de bruit basée sur l'algorithme de Perlin. Cette fonction prend trois arguments en entrée : la position x, la position y et la résolution. La résolution permet de déterminer la taille des variations du bruit et influence la fréquence des motifs générés. En effet, une très grande résolution impliquera des valeurs de bruit quasiment égales pour (x, y) et (x+1, y) par exemple. Et inversement, un bruit très faible entraînera de grands changements entre des coordonnées pourtant proches.
Cette fonction commence par diviser les coordonnées x et y pour les ajuster à l'échelle. Après cela, elle détermine les coordonnées de la cellule de grille dans laquelle se trouve le point (x, y), soit les parties entières de x et y. Grâce à ces parties entières, elle récupère ensuite les parties fractionnaires de x et y. Elles sont utilisées pour représenter la position relative à l'intérieur de la cellule de grille.
Par la suite, les indices des vecteurs de gradient sont calculés en utilisant une permutation de valeurs précalculées basée sur les coordonnées de la grille. Ces indices déterminent les directions des vecteurs de gradient pour chaque coin de la cellule de grille.
Chaque sommet (coin de la cellule) va contribuer à la valeur finale de P (qui est le point de coordonnées (x, y). Plus un vecteur d'un sommet va pointer vers P, plus sa contribution sera grande, donc plus la hauteur de P va augmenter (sur une image, plus la couleur du pixel P sera proche du blanc). A

l'inverse, si un vecteur pointe plutôt à l'opposé, sa contribution sera faible. Deuxièmement, plus P sera proche d'un sommet, plus il sera sensible à la contribution de ce sommet. Cette étape est appelée interpolation.

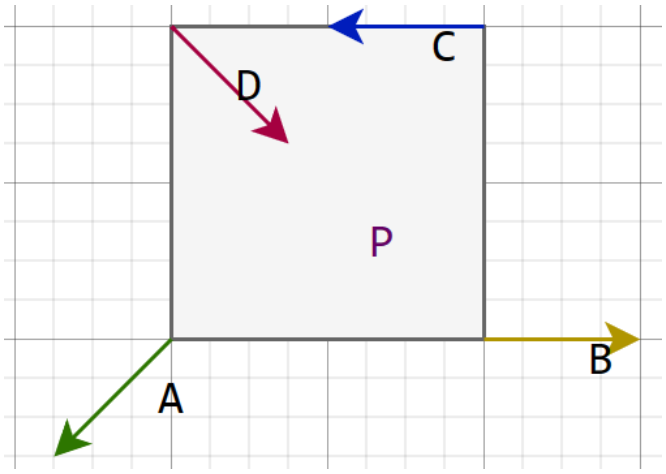


Figure 4 – Schéma représentant les vecteurs obtenus à l'aide de la table de permutation et du tableau de vecteurs unitaires

Les produits scalaires entre les vecteurs de gradient et les vecteurs de décalage (les parties décimale de x et y) sont calculés pour chaque coin de la cellule de grille. Ces produits scalaires représentent les variations locales du bruit.

Les valeurs interpolées sont calculées en utilisant les produits scalaires des coins adjacents et la fonction d'interpolation basée sur la partie fractionnaire Cx. Enfin, la valeur finale du bruit de Perlin est calculée en interpolant linéairement les valeurs le long de l'axe y en utilisant la partie fractionnaire Cy. Cette valeur finale de bruit appartient à l'intervalle [0 ;100] et permet pour chaque case de coordonnées (x, y) dans l'environnement, de déterminer si celle-ci est une case d'un biome désert ou alors d'un biome plaine, affectant alors la probabilité pour un arbre de se trouver sur cette dite case

(très peu de chance dans un désert, évidemment).
Cette attribution des biomes se fait donc directement au moment de la création d'un nouvel environnement avec l'appel à la fonction « new_environment ».
En effet, celle-ci appelle tout d'abord la fonction « fill_environment_biome »
qui va utiliser les valeurs de bruits générées par la fonction « perlin_noise_value ».
Finalement,la fonction « fill_environment_tree » est appelée puisque la génération des arbres nécessite l'assignation des biomes au préalable.

Figure 5 – Capture d'écran d'une génération d'arbres

On peut aisément voir les endroits où les arbres n'apparaissent pas ou peu ; cela correspond aux cases qui font partie de biome désert.

Ici ce sont seulement les biomes qui sont affichés, les cases ' ' étant des déserts, tandis que les cases vides sont des plaines.

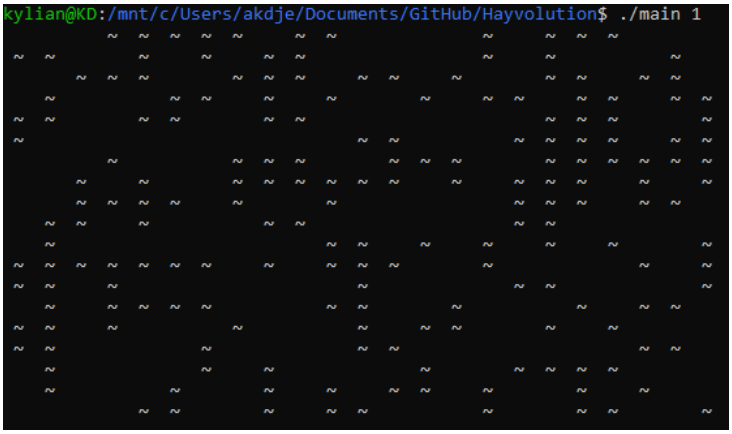


Figure 6 – Capture d'écran d'une génération de biomes

2.2 Animaux

2.2.1 Attribut des animaux

Afin de représenter au mieux les animaux de notre simulation, il leur faut évidemment des attributs. Nous leur avons donné plusieurs caractéristiques différentes qui sont :

- un nom d'espèce pour pouvoir les différencier entre eux.
- une alimentation, pour définir s'ils sont herbivores ou carnivores.
- une vitesse de déplacement, qui définira le nombre de cases qu'ils peuvent se déplacer.
- une perception qui permet à l'animal de voir les objets autour de lui dans un certain rayon.
- une force qui leur sera utile lors du combat.
- un taux de nourriture ainsi qu'une valeur de nourriture totale pour pouvoir récupérer la différence entre les deux.
- un taux de reproduction qui est utile pour savoir si l'animal se repro-

duit ou non en fonction de sa nourriture.
— une liste de fils, qui est utile pour les statistiques ainsi que pour les mutations lors de la reproduction.
— leurs positions dans notre environnement.

Ses caractéristiques sont nécessaires au bon fonctionnement de notre simulation, notamment au niveau de l'interaction entre les différentes espèces.

2.2.2 Mutations et évolution

Le fonctionnement des mutations peut sembler être compliqué, mais notre fonctionnement est plutôt simple. Une mutation est, dans la vie, un changement aléatoire de quelque chose, dans notre simulation, les mutations sont donc des changements aléatoires de certaines caractéristiques de nos êtres vivants. Dans notre évolution, à la fin d'une génération qui correspond à un certain nombre de tours, on teste le taux de nourriture de chaque individu et on doit décider s'ils doivent mourir s'ils possèdent moins d'un certain pourcentage de nourriture, ou bien se reproduire si ce taux se trouve au-dessus de son taux de reproduction dans ses caractéristiques. Les mutations se font à la naissance d'un individu. Il va, par rapport aux caractéristiques de son parent, changer aléatoirement dans un intervalle définissent ses caractéristiques de perception, de vitesse et de force. Les valeurs aléatoires peuvent évidemment être soi négative soit positive. Et sa valeur de nourriture totale sera modifiée en fonction des ses trois caractéristiques, plus le changement de caractéristiques est avantageux, plus il aura besoin de se nourrir car cela lui demandera davantage d'énergie.

2.2.3 Généalogie

Nous avons aussi réfléchi à comment implémenter une structure d'animaux en arbre :

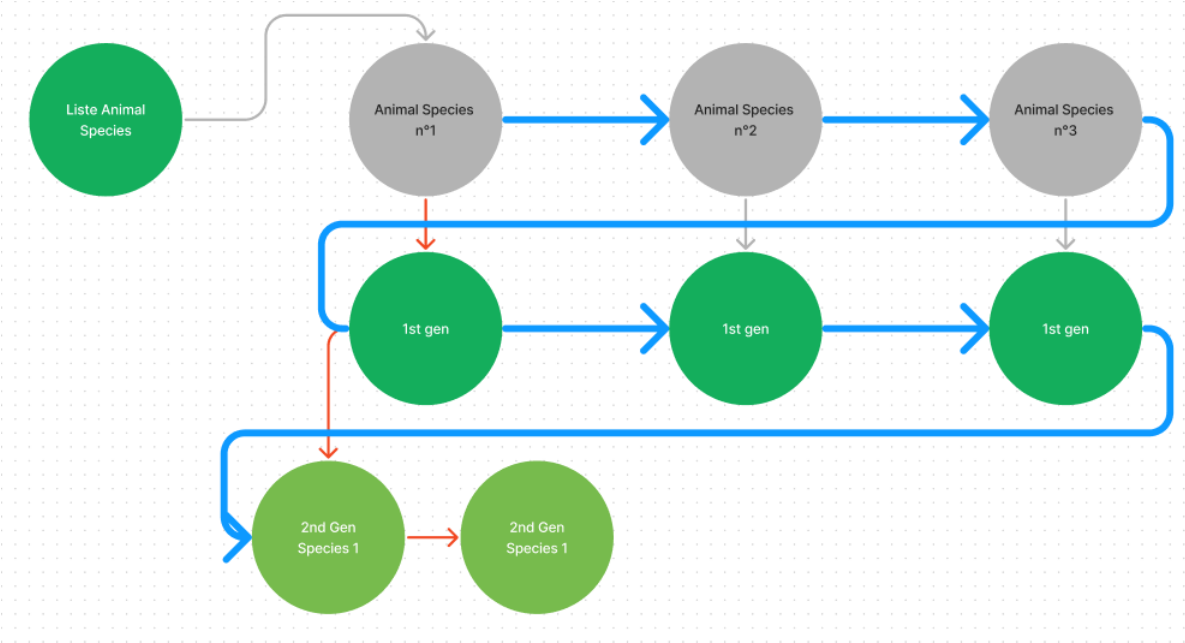


Figure 7 – Représentation en arbre des animaux

Cette représentation a des avantages : elle permet d'avoir un représentant pour chaque espèce qui contient les statistiques de base d'une espèce, et permet d'appeler récursivement des fonctions sur la liste d'espèces. Cette implémentation permet aussi de représenter les liens de parenté entre les animaux ce qui permet d'obtenir des informations cruciales pour après les représenter dans l'onglet statistique. Cependant, cela a un désavantage, tout d'abord, si un animal meurt, on ne peut plus seulement l'enlever de la liste, il faut le garder en mémoire, et on peut lui donner un attribut pour

signifier qu'il est mort pour pallier ce problème.

2.2.4 Interactions

Quand vient le tour d'un animal dans la simulation, un premier test est effectué pour voir si une interaction entre lui et un autre animal ou un élément de décor est possible dans une position adjacente à la sienne. Si ce sont deux animaux, alors on évalue si l'animal qui effectue l'action est un carnivore ainsi que si l'animal attaque est d'une espèce différente de celui qui attaque. Si ces conditions sont remplies, on lance notre fonction de combat qui va en fonction de la force des deux animaux et de la chance, renvoyer si l'animal réussit son attaque ou non. Si c'est le cas, l'animal attaque est donc tué, et est enlevé de l'environnement, et l'animal qui réussit son attaque va pouvoir manger un taux de nourriture proportionnel à la force de l'animal tué.

Ensuite, si l'animal qui effectue l'action est un herbivore et fait face à un carnivore, il va chercher à fuir dans la direction opposée à ce dernier.

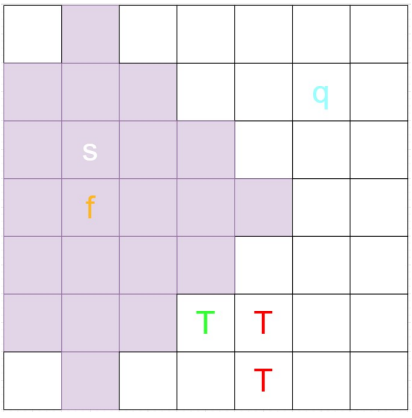


Figure 8 – Intéraction adjacente

Par exemple, sur la figure ci-dessus, l'animal qui effectue l'action est le fox marqué "f" en orange. La zone violette autour de lui est sa perception et nous remarquons qu'un mouton marqué "s" en blanc se trouve à une case adjacente à ce dernier. Le choix de l'interaction ici est donc une action d'attaque. Inversement, si c'était au tour du mouton de jouer, l'action aurait été de fuir dans le sens opposé à celui du renard.

Enfin, si l'animal est un herbivore et qu'il se trouve en face d'un élément de décor, comme un arbre, qui peut être consommé, il va se nourrir et finir son action.

Mais, si le premier test ne détecte aucune interaction dans les cases adjacentes à l'animal, on va chercher à déterminer la prochaine action de notre animal. Auparavant, notre fonction parcourait les cases autour de notre animal en commençant par en haut à gauche puis balayait un rayon autour de lui en fonction de sa perception.

Le problème était que les choix étaient faits sur le moment, c'est-à-dire que, par exemple, un renard qui a en vision un mouton en haut à gauche de lui dans l'environnement bougera directement vers le mouton s'il a faim et ne regardera pas le reste des cases autour de lui. De même, pour un mouton qui apercevait un arbre consommable en haut à gauche ne cherchait pas plus loin et se dirigeait vers ce dernier.

On observait donc au bout d'un moment dans notre simulation une accumulation des êtres vivants dans le coin supérieur gauche de notre en-

vironnement, ce qui est un problème car cela n'est pas très réaliste.

Maintenant, le choix des interactions fonctionne avec un principe de coût de chaque action. Arbitrairement, une attaque (pour un carnivore) ou une fuite (pour un herbivore qui a en vision un prédateur) à un coût de 1. D'autre part, si un herbivore voit un arbre qui peut être mangé, son coût sera de 3.

Ensuite pour chaque action, on ajoute au coût la distance qui sépare l'animal qui effectue son choix d'action et sa destination. Ainsi, dans l'algorithme, le coût minimum de base est fixé à RAND_MAX, qui est la plus grande valeur que l'on peut avoir.

Enfin, si dans sa perception une action possède un coût inférieur à son coût minimum actuel, on met à jour le coût minimum ainsi que la destination de l'animal. Quand toutes les cases visibles par l'animal ont été évaluées, on regarde si le coût est égal à RAND_MAX (c'est-à-dire qu'aucune action valable n'a été trouvé autour de lui), alors sa destination se choisit aléatoirement vers une case où il n'y a pas d'autre animal.

Sinon, grâce à un booléen créer au début de la fonction, on effectue soit une action de déplacement vers la destination choisie, soit une action de fuite à l'opposé de cette destination. Ainsi les mouvements de nos espèces sont maintenant bien plus représentatifs de ce qu'il pourrait se passer dans la vraie vie, nous rapprochant encore un peu plus de notre objectif d'une

simulation la plus proche du réel possible.

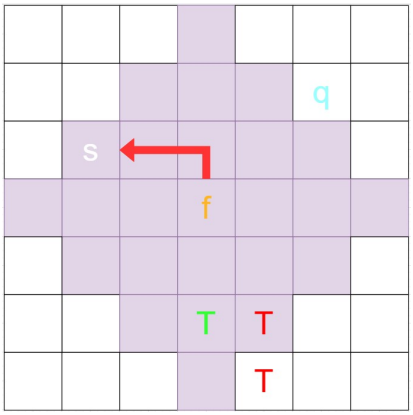


Figure 9 – Choix du renard

On déduit donc que l'action choisit par le renard est celle d'allée vers le mouton afin de l'attaquer (si le renard a faim). Le coût de cette action est donc de $4 = 1 + 3$:

— 1 : Coût de l'action d'attaque

— 3 : distance entre les deux animaux

2.2.5 Déplacement

Un autre point important est la manière dont un animal se déplace. Pour cela il faut prendre plusieurs choses en compte :

— La vitesse de déplacement de l'animal

— Sa destination

1. La vitesse d'un animal dans notre simulation définit le nombre de cases que l'animal pourra parcourir en un déplacement. Nous avons donc implémenté une fonction permettant de calculer le nombre de

déplacement de l'animal en fonction de sa vitesse. Il faut noter que la caractéristique de vitesse est divisée par dix pour trouver le nombre de déplacement garanti, et le reste sera un taux de chance de se déplacer d'une case de plus.

2. Sa destination, elle, est déterminée à l'aide des fonctions de choix d'interactions décrites plus haut.

Auparavant, notre fonction de déplacement fonctionnait ainsi. La première étape de la fonction était de calculer la distance en abscisse et la distance en ordonnée de la position actuelle de l'animal et de sa destination. Elle va ensuite déplacer d'une case ce dernier selon l'axe avec la plus grande différence tout en faisant attention à ne pas dépasser les limites de notre environnement, et également ne pas marcher sur un animal déjà existant. Finalement, elle va répéter ces opérations le nombre de cases souhaitées. Le problème était que cette fonction créait souvent des erreurs de segmentations dû à des étourderies de notre part.

Afin de régler des problèmes que nous trouvions dérangeants, nous avons changé la manière dont les limites de notre environnement sont définies. Précédemment, notre environnement était un simple carré avec pour limite une taille fixe. Maintenant, nous permettons à nos êtres vivants une plus grande variété de mouvements qui leur permet par exemple de ne pas rester bloqués dans un coin et de ne plus jamais en sortir. Ce changement est l'implémentation d'un environnement sans bordure.

Dorénavant, lors de son déplacement, si un animal doit se déplacer en

dehors des limites de l'environnement, il se retrouvera déplacé à la bordure opposée à celle dépassée, permettant ainsi le déplacement sans limites de nos espèces. Évidemment cela fonctionne également lors de la phase de choix d'interaction de chaque être vivant, si sa perception dépasse une bordure, il verra l'autre côté de cette bordure.

2.3 Gestion de la simulation

La simulation se base sur le principe d'arbre de nos animaux. Afin de permettre une meilleure expérience utilisateur, et dans le but de rendre notre simulation plus interactive, nous avons implémenté une fonction permettant à notre utilisateur de créer sa propre espèce pour la voir se développer dans un environnement. Lorsque toutes les espèces qui seront présentes dans la simulation sont définies, on initialise pour chaque espèce un animal de référence dans la liste des animaux, qui ne seront pas présents dans notre simulation. On place ensuite par groupes de 5 maximum le nombre de chaque espèces souhaitées dans l'environnement de manière aléatoire. On lance ensuite notre simulation.

2.3.1 Pendant la simulation

Le parcours de nos animaux pendant le déroulement de la simulation est un parcours largeur de l'arbre ce qui correspond à faire jouer les plus vieilles générations d'abord indistinctement de leur espèce. Dans un élan de respect pour les générations plus âgées. Comme nous l'avions appris avec Mme Chaoued en SUP, un bon parcours largeur doit utiliser une queue. C'est donc avec ce principe que nous parcourons nos animaux. Un

premier passage est effectué afin de ne pas prendre en compte les espèces de référence de notre simulation. Ensuite, on fait jouer chaque animal dans son ordre de passage : Tout d'abord, on regarde si l'animal est en vie puisque dans notre implémentation, nous nous devons de garder tout les animaux, même ceux qui sont morts, dans notre liste.
Ensuite, on réduit sa nourriture proportionnellement à sa force et on lance enfin les différentes fonctions d'interactions.

2.3.2 Fin d'une génération

Lorsqu'on atteint une fin de génération, on appelle une nouvelle fonction qui, comme celle-ci utilisée pendant la génération, va parcourir les animaux avec un parcours largeur en ignorant les espèces références. Puis lancera notre fonction testant le taux de nourriture de chaque animal afin de savoir si cet animal meurt ou se reproduit.

2.3.3 Équilibrage

Pour ce qui est de l'équilibrage, nous avons dû effectuer de nombreux tests avec les trois espèces animales présentes par défaut dans la simulation (les renards, les moutons et les écureuils). Il fallait créer des espèces assez distinctes au niveau des statistiques, ce qui garantirait une évolution qui soit différente pour chaque espèce au sein d'une même simulation, tout en s'assurant que d'une simulation à l'autre, les résultats puissent être différents. Ainsi, nous avons dû faire en sorte qu'aucune espèce ne soit trop avantagée ou trop faible ce qui aurait entraîné des simulations avec des progressions et des finalités quasiment identiques. Cependant, les arbres ont aussi dû être modifiés plusieurs fois en ce qui concerne la fréquence à

laquelle ils pouvaient à nouveau être mangés, ainsi que la valeur de nourriture rendue à un animal s'y nourrissant. Cet équilibrage sera évidemment modifié lorsque l'utilisateur décide de créer une nouvelle espèce, en plus des trois d'origine, au début d'une simulation.

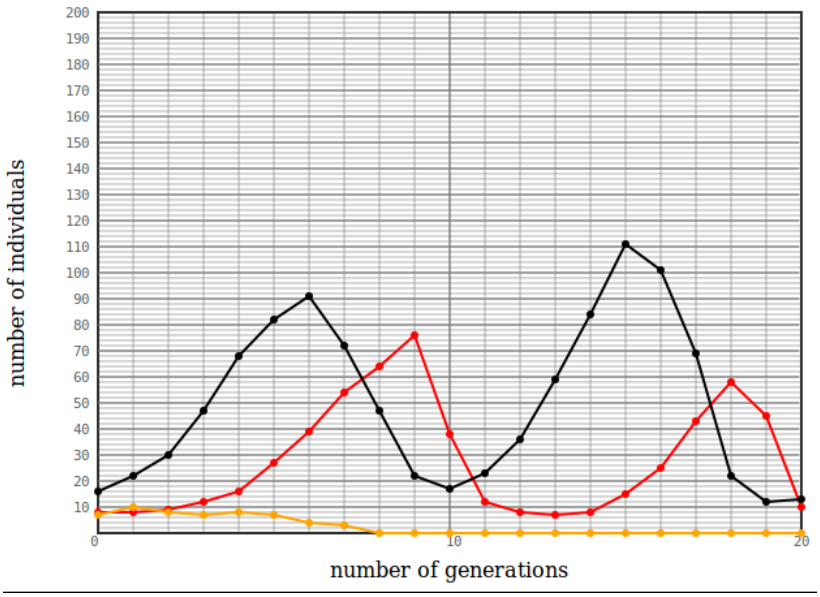


Figure 10 – Caption

/

2.4 Statistiques

Garder en mémoire des statistiques sur ce qu'il se passe dans la simulation est crucial. Comme nous l'avons vu dans la partie précédente, l'affichage basique sous forme de caractère permet d'avoir un aperçu des interactions entre les animaux, ainsi que de leurs positions et déplacements. Cependant, l'affichage est rafraichie rapidement et il peut vite devenir lassant de le regarder. De plus, il est difficile d'estimer le nombre d'individus en regardant simplement l'environnement. Comme l'un des objectifs de ce projet est de pouvoir observer l'évolution et les mutations chez une espèce, nous devons aussi trouver un moyen de rendre compte de l'évolution des attributs moyens d'une espèce.
Lorsque l'interface n'était encore qu'en ligne de commande, nous utilisions une fonction printf() pour afficher les statistiques moyennes de chaque espèce à chaque génération. Cette façon de restituer les statistiques était très limitée car une fois une génération passée, nous n'avions plus accès à celles de la génération précédente, et en plus ne permettait pas de visualiser rapidement l'évolution.
Pour régler ce problème, nous avons utilisé une librairie appelée GLG qui elle même utilise GTK/GDK. Cette librairie nous permet d'afficher des graphiques sur l'évolution au cours du temps de caractéristiques spécifiques. Nous détaillerons l'implémentation de la fenêtre des graphes dans la partie suivante, ici nous allons nous intéresser aux statistiques que nous avons choisi de suivre, comment et pourquoi.

2.4.1 Nombre d'individus et caractéristiques moyennes

Dans un premier temps, nous avons créé une struct Stats, au moment où l'utilisateur lance la simulation avec un nombre d'espèces qu'il a choisi, on crée des tableaux d'entiers qui vont stocker les attributs suivants.

- char_species : qui permet de garder en mémoire à quel index correspond une espèce dans les tableaux.
- species_number : contient le nombre d'individus en vie pour une espèce à une case i.
- new_born : contient le nombre d'individus nés à la dernière génération.
- died : contient le nombre d'individus morts à la dernière génération.
- x_strength : contient la moyenne de la statistique force pour une espèce.
- x_speed : contient la moyenne de la statistique vitesse pour une espèce.
- x_sense : contient la moyenne de la statistique perception pour une espèce.
- x_totalfood : contient la moyenne de la statistique nourriture totale pour une espèce.

Comme l'odre de jeu dans la simulation n'est pas choisi par espèce mais pas ancienneté dans la simulation, nous avons réalisé la fonction species_index(Animal* animal, Stats* stats) qui lorsque l'on traite un animal va comparer son attribut species (un char) avec les char dans le tableau char_species pour renvoyer l'index où se trouve l'espèces.

Figure 11 – Dans cet exemple, lorsque l'on traite un mouton la fonction va renvoyer 0

Comme les animaux peuvent être mangés ou mourrir de faim avant la fin d'une génération, nous avons fait le choix de mettre à jour ces statistiques en temps réel. Lorsque l'on appelle la fonction kill_animal() par exemple, on va décrémenter species_number et incrémenter died de 1. Nous faisons de même lorsqu'à la fin d'une génération nous allons appeller la fonction life_or_death(), qui décide si un animal meurt ou se reproduit.

Pour mettre à jour les statistiques x_strength, x_speed, x_sense et x_totalfood nous allons à l'inverse faire comme nous faisions avant, c'est à dire que l'on va effectuer un parcours profondeur sur chaque espèce d'animal, faire la somme totale d'une caractéristiques puis la diviser par le nombre total d'individus de cette espèce (contenu dans species_number([i])) pour pouvoir mettre à jour nos moyennes.

2.4.2 Affichage des statistiques

Toutes ces statistiques seraient illisibles si elles étaient affichées en même temps, c'est pourquoi nous avons utilisé un bouton qui permet de choisir si l'on souhaite avoir l'aperçu global des statistiques, c'est à dire le total d'individus par espèce, ou bien pour une espèce si l'on souhaite suivre le nombre de mort et de nouveaux nés, ou bien l'évolution des caractéristiques strenght, speed, sense et totalfood.
Nous détaillerons les détails de l'affichage dans la partie suivante.

2.5 Interface

Au lieu de simplement afficher l'environnement et ses animaux dans le terminal, nous avons finalement réalisé l'interface avec GTK.

2.5.1 Interface et boutons

Dans la fenêtre principale nous affichons donc la board dans un GTK container, dans ce container il y a un widget GTK grid des dimensions de notre environnement, et chaque case de la grille constitue un GTK label, ensuite à chaque round, nous lisons la matrice de l'environnement et mettre à jour les GTK label dans chaque case.

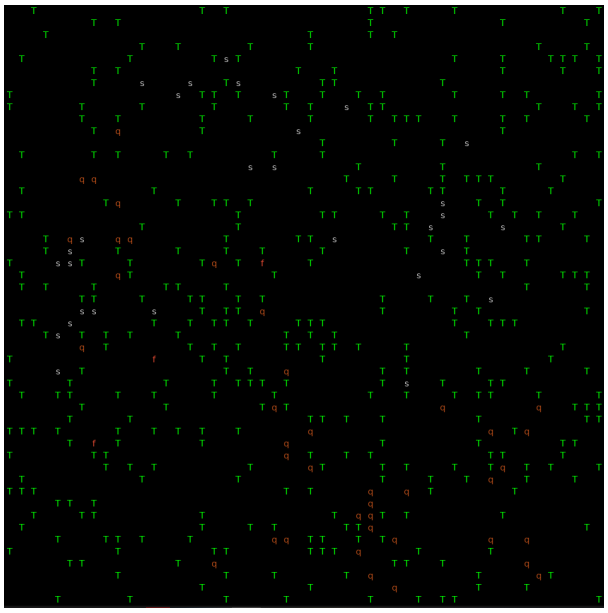


Figure 12 – Le print de la matrice dans l'interface

Nous avons ensuite intégré à cette fenêtre les boutons qui avaient été

programmés par Lino :

— Un bouton "QUIT" qui permet de quitter la simulation.

— Un bouton "STATISTICS" qui permet d'ouvrir la fenêtre des graphes.

— Un bouton "PAUSE" qui devient un bouton "RESUME" lorsque la simulation est en pause.

2.5.2 Fenêtre graphes

Grace à la librairie GLG nous avons pu créer une fenêtre dédiée à afficher les graphiques des statistiques que l'on lui transmet. Comme expliqué dans la partie sur les statistiques, l'utilisateur peut choisir entre afficher les statistiques globales ou bien spécifiques pour chaque espèce.

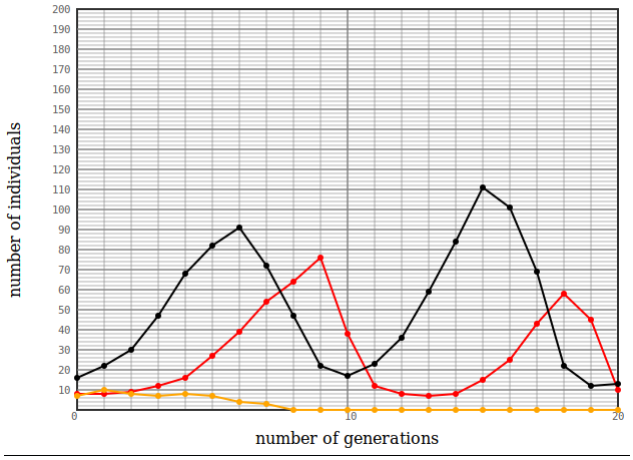


Figure 13 – Les statistiques globales (nombre d'individus par espèce)

2.5.3 Ajout d'espèces

Nous avons aussi intégré la possibilité de rajouter une ou plusieurs espèces dans la simulation avant de lancer la simulation. Cette fonctionnalité permet de laisser plus de possibilités à l'utilisateur. En effet, il a complètement le choix des statistiques qu'il va donner à l'espèce qu'il crée même si elle risque de ne pas être équilibrée.

2.6 Site web

Le travail sur le site web est désormais terminé. Nous sommes fiers d'annoncer que le site web est complet et, représentant une facette importante dans la réalisation de notre projet puisqu'il en est la vitrine. Bien que nous ayons encore la possibilité d'apporter des améliorations esthétiques, nous avons rajouté du contenu sur le site site. Actuellement, il propose plusieurs pages clés qui fournissent aux utilisateurs les informations essentielles sur notre projet.

Tout d'abord, nous avons une page de présentation du projet qui met en évidence les objectifs, les fonctionnalités et les résultats attendus. Cette page offre un aperçu global du projet, permettant aux visiteurs de comprendre rapidement son importance et son utilité.

Ensuite, nous avons ajouté une page dédiée au téléchargement du projet. Cette page offre aux utilisateurs la possibilité d'accéder directement à notre projet, ainsi qu'aux rapports de soutenance correspondants. Cela

facilite l'accès aux ressources nécessaires pour comprendre et utiliser notre travail.

De plus, nous avons inclus un lien vers l'organisation GitHub de Kamels, permettant aux utilisateurs d'accéder à notre dépôt de code source.

Une autre page importante que nous avons créée est celle de présentation du groupe. Cette page met en valeur les membres de notre équipe, leurs compétences et leurs contributions individuelles au projet. Cela permet aux visiteurs de connaître les personnes derrière le projet et de reconnaître l'effort collectif qui a été déployé.

Enfin, nous avons développé une page de documentation dédiée. Cette page fournira toutes les spécifications nécessaires pour utiliser notre projet de manière optimale. Nous y inclurons des instructions détaillées pour obtenir des résultats intéressants. Cette documentation jouera un rôle crucial pour faciliter l'adoption de notre projet par d'autres utilisateurs qui ne savent pas forcément programmer.

En conclusion, nous avons réussi à créer un site web complet, fournissant des informations claires et pertinentes sur notre projet.

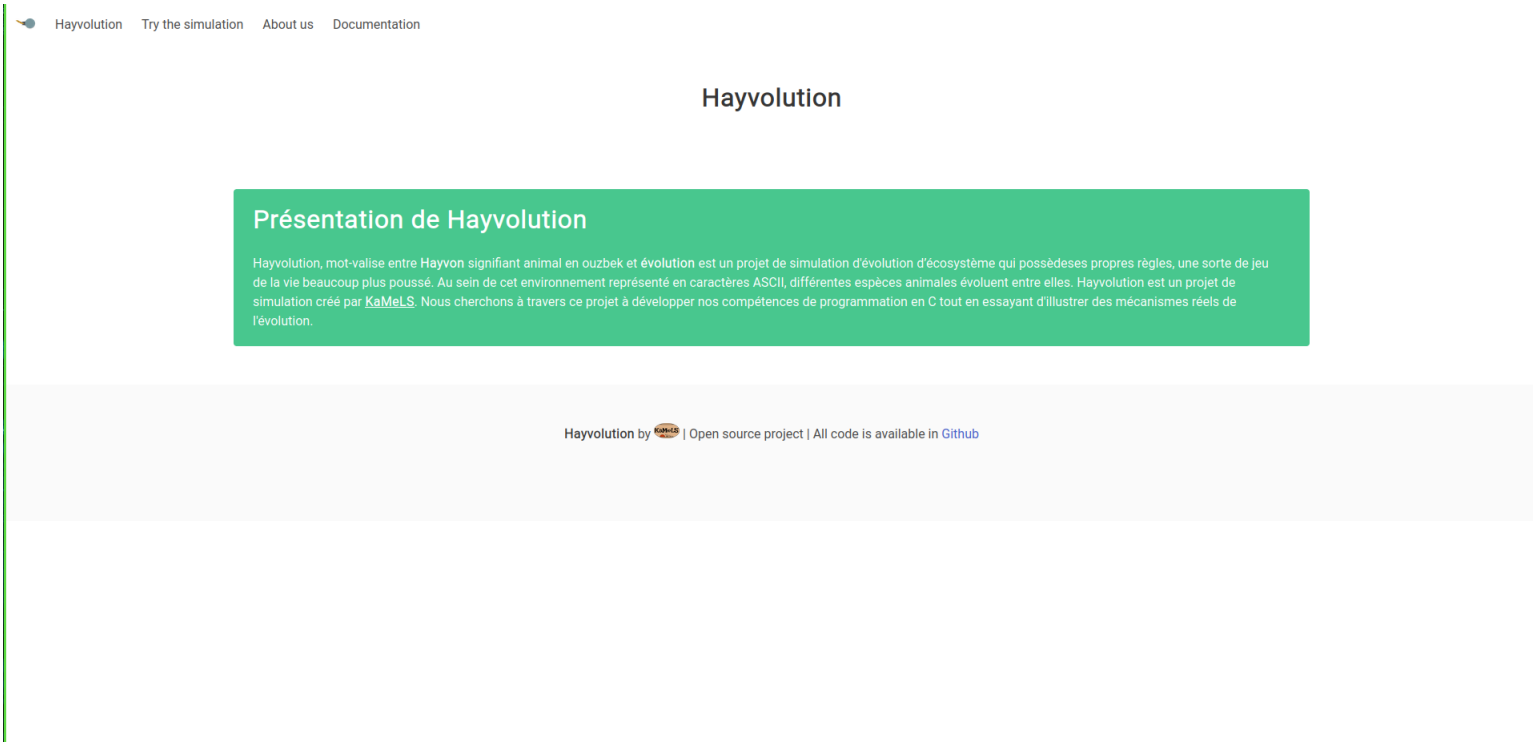


Figure 14 – La page de présentation du projet sur le site web

3 Conclusion

3.1 Avancement atteint

3.1.1 Comparaison avec notre idée initiale

Tâche	Avancement atteint
Interface	80%
Statistiques	90%
Animaux	100%
Environnement	90%
Interactions	100%

Nous avons globalement atteint notre objectif et nous sommes contents de notre projet. Toutes les fonctionnalités que nous avions initialement prévues ont été implémentées dans la simulation. Néanmoins, nous devons reconnaître que nous aurions pu pousser la complexité de la simulation en rajoutant d'autres éléments à l'environnement, comme de l'eau par exemple. Cependant, en faisant cela nous risquions de créer de nouveaux déséquilibres dans la simulation d'autant plus que l'équilibrage est déjà assez délicat à ce stade.

Au niveau de l'interface, nous aurions pu rajouter encore plus de fonctionnalités et de possibilités de personnalisation. Pour ce qui est des statistiques, nous aurions pu en reccueillir encore plus, surtout comme nous y avions pensé de dissocier certaines branches d'une espèce lorsqu'elles avaient évolués trop différemment. Pour ce qui est de l'affichage des sta-

tistiques, la librairie glg nous a permis de pouvoir dessiner des graphes rapidement mais nous n'avons pas su exploiter toutes ses fonctionnalités.

3.1.2 Rajouts et abandons d'idées

Pour les animaux nous avions vaguement évoqué l'idée d'un système d'intelligence artificielle afin de gérer leurs choix de déplacements, cependant, ce projet était irréalisable, beaucoup trop complexe à implémenter pour chaque individu de chaque espèce.
De plus comme nous avons accès à la généalogie d'une espèce, nous avions pensé à exploiter cette donnée couplée aux statistiques pour reproduire le mécanisme de divergence évolutive, c'est à dire que si une branche de l'espèce est devenue trop différente d'une autre, on considère qu'elles forme deux espèces à part entière. Cette idée bien qu'intéressante, nous a semblée trop coûteuse à mettre en place donc nous avons préféré l'abbandonner.
En ce qui concerne l'environnement, nous aurions voulu incorporer au sein du décor de l'eau, en plus des arbres, ainsi qu'une jauge de soif pour tous les animaux, ainsi les carnivores auraient aussi également été dépendants de la génération du terrain. Nous avions également envisagé de possibles obstacles infranchissables tels que des montagnes pour séparer totalement différentes parties de la carte, mais cela aurait été intéressant avec la création d'environnements bien plus grands.

3.2 Difficultés rencontrés

Au fil de l'avancée de notre projet, nous avons créé de nombreuses classes différentes tels que Environment, Animal ou Tree, chacune de ces classes étant définie dans un header particulier relatif au fichier .c où la classe était le plus utilisé (Environment dans generation.h par exemple). Cependant, pratiquement tous nos fichiers .c utilisent au moins une fois une classe définie dans un header différent que le .h qui lui est associé. N'arrivant pas à include chaque header depuis les fichiers .c sans causer d'erreurs, nous avons simplement créé un header "class.h" regroupant l'ensemble des classes de notre programme. Comme expliqué plus haut dans la partie sur le déplacement des animaux, la fonction permettant ce mouvement nous a causé de nombreux problèmes menant inévitablement à des erreurs de segmentations. Nous avons passé plusieurs heures sur ce problème qui parraissait simple pour au final avoir dû coder de manière brute chaque élément. Au final avec l'implémentation de notre environnement sans bordure, les parties de codes brutes ont pu être remplacées par des fonctions plus générales, plus propres et plus lisibles. Pour la fonction de bruit, nous avons rencontré des problèmes pour obtenir des résultats satisfaisants en terme de cohérence des biomes. Les limites entre ceux-ci n'étant parfois pas assez prononcé, et ce qu'importe la résolution appliqué à l'algorithme, menant à une génération d'arbres qui ne permettaient pas réellement de visualiser cette différenciation de biomes.

3.3 Améliorations possibles

Pour l'environnement, il est possible d'envisager une plus grande liberté de personnalisation offerte à l'utilisateur. Lui permettre de biaiser la génération de l'environnement pour vouloir plus ou moins d'arbres, des biomes plus ou moins grands ou qu'un seul biome en particulier par exemple.

Pour les animaux, il est possible d'envisager un meilleur suivi des statistiques pour savoir les caractéristiques moyennes de chaque génération à chaque instant de la simulation pour pouvoir suivre les mutations de chaque nouvelle générations et sa réussite dans l'environnement.
Enfin, comme dit précédemment, nous pouvons encore améliorer l'interface et rajouter des fonctionnalités et des possibilités de personnalisation de la simulation.

3.4 Conclusion

Ce projet S4 a été une expérience enrichissante pour tous les membres du groupe qui nous a permis de développer nos compétences techniques, de renforcer notre collaboration en équipe et de découvrir de nouvelles approches pour résoudre des problèmes complexes. Au fil de ce rapport final, nous souhaitons mettre en évidence les accomplissements que nous avons réalisés tout au long de ce projet, ainsi que les enseignements que nous en avons tirés.

Tout d'abord, conformément à notre idée de base que nous avons présenté en introduction, nous sommes fiers d'annoncer que nous avons atteint notre objectif initial. Malgré les obstacles et les défis rencontrés en cours de route, nous avons su faire preuve de persévérance et d'engagement pour maintenir le cap et progresser vers la réalisation de notre projet. Cela démontre notre capacité à gérer efficacement notre temps et à nous adapter aux situations changeantes, tout en maintenant notre motivation intacte.

L'une des réalisations majeures de ce projet a été le développement d'un produit final abouti et pertinent. Grâce à notre travail d'équipe et à notre collaboration étroite, nous avons réussi à concevoir, à programmer et à mettre en œuvre une simulation cohérente fonctionnelle qui répond aux attentes spécifiques du projet. Nous avons intégré les fonctionnalités clés demandées, en les adaptant et en les optimisant au fur et à mesure de notre progression. Cette réussite témoigne de nos compétences en matière de développement logiciel et de notre capacité à concrétiser une vision commune.

En outre, ce projet nous a également permis d'améliorer nos compétences en matière de gestion de projet. Nous avons appris à organiser efficacement

les tâches, à établir des échéances réalistes et à répartir équitablement la charge de travail entre les membres de l'équipe. La communication régulière et transparente a été essentielle pour assurer la coordination et la cohésion de l'équipe. Nous avons également développé notre capacité à résoudre les problèmes de manière collaborative, en identifiant les obstacles et en proposant des solutions alternatives et viables pour les surmonter.

Figure 15 – KaMeLS at the zoo