HEIG-VD — POA

Laboratoire 4 – Rapport

[redacted]

29 juin 2024

1 Introduction

L'objectif de ce projet est de développer une application de simulation où Buffy, la tueuse de vampires, les humains et les vampires, collectivement appelés "humanoïdes", interagissent dans un environnement en mode tour par tour. Le but de cette simulation est d'analyser les dynamiques de survie des humains et l'efficacité de Buffy à les protéger contre la menace des vampires. L'application offre une plateforme pour conduire et observer plusieurs itérations de ces interactions, générant ainsi des insights statistiques sur le taux de succès de Buffy dans divers scénarios.

Dans cette simulation, les humanoïdes sont placés aléatoirement sur une grille dont la taille, ainsi que le nombre d'humains et de vampires, sont prédéterminés par l'utilisateur. À chaque tour, les humanoïdes décident et exécutent leurs actions simultanément pour assurer une équité dans la séquence des actions. Le rôle principal de Buffy est d'attaquer et d'éliminer les vampires, tandis que les vampires ciblent les humains avec l'intention de les tuer ou de les transformer en vampires. Les humains se déplacent aléatoirement sur la grille. L'application comprend également un menu permettant aux utilisateurs de passer des tours, de visualiser des statistiques ou de quitter le programme. Pour l'analyse statistique, la simulation exécute 10000 itérations pour déterminer le pourcentage de scénarios où Buffy réussit à protéger au moins un humain.

La mise en œuvre implique une conception basée sur des classes pour séparer la gestion de l'interface graphique du comportement des humanoïdes. Chaque classe d'humanoïde est liée à un objet `Action` qui dicte son prochain mouvement, tandis que la classe `Field` gère la grille et coordonne les interactions entre les humanoïdes. Cette approche assure une modularité et permet à la simulation de fonctionner avec ou sans affichage graphique. Ce rapport détaillera la structure des classes, les choix de conception, ainsi que les résultats statistiques.

2 Instructions de compilation

Nous avons conservé la structure avec *cmake* pour notre implémentation. Nos versions de la suite de compilation utilisée par *cmake* sont les suivantes :

- **cmake** : 3.29.5
- **ninja** : 1.12.1
- **gcc** : 14.1.1 20240522
- Compilé sur Linux avec le kernel 6.9.4

À noter que nous utilisons dans ce laboratoire deux fonctionnalités introduites avec C++ 23, nécessitant donc une version plus ou moins récente de gcc.

3 Modélisation

La modélisation du problème est donnée dans la figure 1 en annexe.

4 Choix et hypothèses de travail

Une modélisation classique a été choisie pour ce projet, cependant certains choix ont dû être faits pour permettre la factorisation du code que nous allons détailler dans les sections à venir.

4.1 Utilisation des smart pointers

Un aspect primordial de notre modélisation a été de décider d'utiliser les pointeurs intelligents pour l'ensemble des structures de données créées. Ce choix nous a semblé pertinent, car certaines opérations, comme la transformation, nécessiteront de créer dynamiquement une nouvelle instance d'un humanoïde tout en évitant le *slicing*. Avec les pointeurs intelligents, nous avons donc pu permettre la création dynamique de ces instances partagées sans risque de fuites de mémoires en cas d'erreur ou interruption spontanée. Cette décision a néanmoins requis une plus haute attention quant à la gestion des références sur `this` dans les classes héritant de `Humanoid`, via l'utilisation du mécanisme `shared_from_this()` pour éviter des libérations doubles des pointeurs intelligents sur les humanoïdes.

Ainsi, toutes les structures nécessitant une référence vers un objet de la hiérarchie des humanoïdes devra utiliser des pointeurs intelligents.

4.2 Hiérarchie des humanoïdes

L'héritage des types d'humanoïdes est assez classique, chaque type hérite d'une classe abstraite commune `Humanoid` qui permet de gérer la position et donne les méthodes nécessaires pour la gestion des actions. Il est néanmoins important de préciser que nous avons décidé de ne pas faire hériter `Buffy` de `Human` malgré son comportement par défaut similaire, car elle ne peut pas être tuée ni poursuivie par les vampires. Quoi qu'il en soit, nous n'aurions de toute façon pas pu classifier `Buffy` comme étant un humain, car elle prend simplement le rôle d'un humain quand son travail de chasse est terminé.

Quand le tour est à l'étape de détermination des actions, `Field` va appeler la méthode `createAction` de `Humanoid` qui va simplement faire appel à `getNextAction` qui est une méthode *pure-virtual*. Cette dernière va donc retourner un nouveau pointeur unique sur l'action qui sera stocké dans le champ correspondant par `createAction`. Cela permet d'extraire la génération dans une méthode spécifique sans briser l'encapsulation en ajoutant un setter vers ce champ qui devrait rester réservé à la classe `Humanoid`.

Le pointeur est unique vu que son utilisation est réservée à la classe `Humanoid`, notamment lors de l'appel à `executeAction` par le `Field` lorsque tous les humanoïdes auront généré leur action suivante. Un humanoïde pourra donc, au moyen des deux méthodes plubliques correspondantes, être tué ou déplacé lors du passage par `executeAction` de sa propre instance, ou l'instance de son chasseur.

4.3 Hiérarchie des actions

Toutes les actions héritent d'une classe abstraite nommée sans trop de surprises `Action`. Celle-ci stocke le sujet de l'action, que l'on définira plus proprement ci-après, sous la forme d'un pointeur faible sur un `Humanoid`. Cela évite de croiser les références entre les humanoïdes et les actions, car nous partons du principe qu'un humanoïde est le *owner* de sa prochaine action via le pointeur unique, donc une action sera uniquement un observateur de l'humanoïde sujet et ne devrait jamais y être propriétaire, car l'action doit pouvoir être libérée même si l'humanoïde sujet ne l'est pas.

Une action sera toujours exécutée en différé, via sa méthode `execute` prenant le champ de jeu en paramètre, l'exécution est définie de manière abstraite dans cette classe de base.

4.3.1 Action d'attaque

Cette action est la plus basique de l'hiérarchie mise en place pour les règles énoncées. Définie dans `Kill`, le sujet sera l'humanoïde tué et celui-ci recevra simplement un appel à sa méthode `kill` quand l'action sera exécutée.

4.3.2 Action de transformation

Cette action est une spécialisation de l'action d'attaque définie dans `Transform`, car un vampire va cibler un humain qui sera forcément tué. Il y a néanmoins une chance sur deux, déterminé via la gestion aléatoire décrite ci-après, que cet humain soit également transformé en vampire, qui sera ajouté dans le champ de jeu en cas de succès de l'expérience de Bernoulli.

Les vampires sont donc les seuls qui utilisent cette action. Nous avons décidé de faire l'expérience de Bernoulli dans l'action plutôt que dans le vampire, car cela nous semblait plus cohérent que ce soit la transformation qui peut potentiellement échouer plutôt que ce soit le vampire qui détermine si son action va transformer ou simplement tuer l'humain.

4.3.3 Action de mouvement

`Move` est l'action de déplacement qui est utilisée par tous les humanoïdes, celle-ci peut néanmoins être selon un déplacement aléatoire ou ciblé. Le sujet de cette dernière sera évidemment l'humanoïde en déplacement.

Cette action définit deux champs supplémentaires par rapport aux deux autres : `range` et `target`. Le champ `range` détermine le nombre de déplacements effectués durant le même tour, pour les déplacements de Buffy qui, en cas de chasse, se font deux cases à la fois. Le champ `target` est modélisé par un optional vers une position qui, si défini, orientera le mouvement dans sa direction. Dans le cas où `target` est vide, un mouvement aléatoire dans le champ de jeu sera choisi.

4.3.4 Gestion de l'aléatoire

Pour gérer l'aléatoire des actions de transformation et de déplacement, nous avons défini un générateur de type *Mersenne Twister* dans le champ de jeu. Il est initialisé selon une seed non déterministe à chaque initialisation d'un champ de jeu. Cela permet, dans un déroulement normal, d'assurer que les actions soient aléatoires en restant dans un ordre prévisible pour des cas de tests. Pour les simulations, le champ de jeu étant initialisé à chaque itération, il est garanti que chaque exécution et chaque itération suivra un ordre de décision différent.

Dans nos cas de tests, nous avons par contre pu utiliser une seed fixe pour le générateur, permettant de savoir de façon déterministe si un changement effectué était équivalent, améliorant, ou détériorant. Étant donné l'ordonnancement fixe des actions, cela nous a garanti un environnement stable pour effectuer les tests sans avoir à faire recours à une part de hasard.

4.4 Coordination des actions

Le champ de jeu mentionné ci-avant, implémenté par la classe `Level`, permet de coordonner tous les humanoïdes et leurs actions de façon indépendante à l'affichage qui sera géré par le contrôleur décrit plus loin. Notre implémentation stocke donc la taille de la grille, le numéro de tour, le générateur aléatoire et la liste des pointeurs partagés d'humanoïdes. Le placement des humanoïdes est effectué dans le constructeur, prenant en plus le nombre d'humains et le nombre de vampires désirés, ils seront placés aléatoirement dans la grille. Il est important de noter que plusieurs humanoïdes peuvent être placés sur la même cellule.

En plus d'une méthode permettant d'ajouter un humanoïde après la construction (`addHumanoid`, nécessaire pour la transformation) nous avons défini deux méthodes génériques `findClosest` et `getCount` qui utilisent le principe de RTTI pour obtenir, respectivement, l'humanoïde de type T le plus proche et le compte de tous les humanoïdes de type T. Ces deux méthodes sont donc génériques et implémentées dans un en-tête séparé pour nous permettre d'utiliser les `dynamic_pointer_cast` qui sont obligatoires dans ce contexte pour filtrer selon ce qui est nécessaire en permettant également, par souci d'évolutivité, de détecter des sous-classes éventuelles des types d'humanoïdes.

La méthode `getCount` fonctionne de la même manière et permettra au contrôleur de déterminer si une partie pour la simulation a été gagnée par Buffy ou non. Nous avons décidé de procéder ainsi, car ce sont deux valeurs calculées qui ne sont pas extrêmement coûteuses et évitent de devoir ajouter des événements à appeler manuellement en cas d'attaque d'un humain, vampire ou transformation d'un humain en vampire.

4.5 Interface utilisateur

Le contrôleur, implémenté par la classe `Controller`, va s'occuper de récupérer les inputs de l'utilisateur, vérifier la bonne validité de ceux-ci et gérer un champ de jeu pour le déroulement étape-par-étape. Il ajoute en outre un mécanisme de simulation qui peut être déclenché par l'utilisateur pour effectuer une certaine

quantité d'itération et afficher le taux de réussite à l'utilisateur. Nous avons tenu à ce que le contrôleur ne gère aucune règle de gestion du jeu, ni la gestion de la logique du jeu, car ceux-ci doivent être entièrement gérés par le champ de jeu pour permettre des simulations réalistes.

Le constructeur du contrôleur prend en paramètres les valeurs qui seront utilisées pour initialiser les champs de jeux du déroulement étape-par-étape et des simulations.

Le programme peut ensuite appeler la méthode `run` pour démarrer le jeu, cette méthode est bloquante. En cas de signal nécessitant un arrêt soudain du programme, il est toujours possible d'appeler `quit` qui arrêtera le déroulement de la partie.

Après chaque action, l'utilisateur peut entrer une commande qui est définie dans une `std::map`. Cette map contient pour chaque identifiant de commande une description et une action correspondante à effectuer sur le contrôleur. Elle permet en outre d'être flexible au cas où d'autres commandes serait nécessaire, par exemple, avec l'ajout de méthodes pour ajouter des humanoïdes pour effectuer certains tests.

Une méthode générique `handleCommand` va s'occuper de trouver la commande demandée par l'utilisateur, et appeler son action. Elle affichera un message d'erreur le cas échéant.

Le déroulement étape-par-étape ne s'arrête jamais, on pourra donc y observer le comportement de déplacement aléatoire effectué par Buffy si tous les vampires ont été tués. L'affichage du champ de jeu est abstrait dans une méthode statique de la classe `FieldDisplayer`, définie ainsi pour éventuellement permettre une gestion différente de l'affichage des symboles selon les plateformes, par exemple dans le cas où nous aurions voulu ajouter des couleurs aux symboles.

Lors des simulations, le champ de jeu continue à évoluer tant qu'il reste des vampires. Une fois tous les vampires tués, s'il reste encore des humains à la fin, alors l'itération est considérée comme une victoire. Le taux de réussite sera ensuite affiché à l'utilisateur.

5 Résultats des statistiques

Avec la modélisation décrite ci-avant, nous avons lancé 20 threads en parallèle et effectué une moyenne sur 10 séries de 10'000 itérations par thread. Les conditions de la grille étaient telles que décrites dans la donnée : taille de 50 par 50, 10 vampires et 20 humains.

Nous obtenons donc une moyenne globale à **44.37%** ($\sigma = 0.339$).

6 Tests effectués

Pour ce laboratoire, nous avons effectué plusieurs tests manuellement en utilisant des conditions de départ différentes pour valider le bon comportement des actions demandées. Nous avons aussi testé la reproductibilité d'une solution en ajoutant manuellement une seed pour la génération des nombres aléatoires.

Les tests de comportement ont été effectués dans un champ de jeu de taille réduite afin de faciliter l'observation et l'analyse des interactions entre les différentes entités (humains, vampires, et Buffy). Cette approche permet de vérifier plus précisément le comportement attendu de chaque entité dans un espace contrôlé.

Nous avons donc fait l'hypothèse que ces tests seront suffisamment complets pour valider le bon fonctionnement de l'ensemble de nos structures de données, le résumé est donné en table 1 en annexe.

7 Conclusion

Ce laboratoire nous a permis de mettre en pratique de nombreux concepts avancés de la programmation orientée objet en C++. Nous avons utilisé des pointeurs intelligents pour gérer la mémoire de manière sûre et efficace, en évitant les fuites de mémoire. L'héritage a été employé pour structurer notre hiérarchie de classes, permettant une conception modulaire et extensible. Nous avons également utilisé des fonctions anonymes pour simplifier la gestion des commandes, rendant notre code facilement maintenable.

Description	Résultat attendu et observé
Arguments	
Exécution du programme avec des paramètres de taille, nombre d'humains et de vampires spécifiés	Le champ est généré avec le nombre indiqué d'humains et de vampires, ainsi qu'une Buffy. Toutes les créatures sont positionnées aléatoirement.
Exécution du programme avec des arguments invalides (par exemple : arguments manquants, valeurs < 0, valeurs non numériques)	Un message d'erreur s'affiche et le programme se termine.
Exécution du programme avec des arguments valides mais contenant des valeurs égales à 0	Un message d'erreur s'affiche et le programme se termine.
Commandes	
L'utilisateur entre la commande « n »	L'affichage se met à jour, chaque créature de la simulation exécute une action.
L'utilisateur appuie sur la touche Entrée	L'affichage se met à jour, chaque créature de la simulation exécute une action.
L'utilisateur entre la commande « q »	Le programme se ferme correctement, toutes les allocations dynamiques sont libérées.
L'utilisateur entre la commande « s »	Une simulation fixe (10'000 cycles) commence avec les paramètres initiaux. À la fin, le pourcentage de victoires de Buffy est affiché.
Comportement	
Buffy se trouve à plus d'une case du vampire le plus proche	Buffy se déplace de deux cases vers le vampire le plus proche à chaque tour.
Buffy se trouve à moins d'une case du vampire le plus proche	Buffy tue le vampire sans se déplacer.
Un vampire se trouve à plus d'une case de l'humain le plus proche	Le vampire se rapproche d'une case de l'humain le plus proche.
Un vampire se trouve à une case ou moins d'un humain	Le vampire attaque l'humain et soit le transforme en vampire soit le tue, sans se déplacer.
Un humain passe un tour de simulation	L'humain se déplace d'une case de manière aléatoire tout en restant dans le champ de jeu.
Aucun humain restant en vie	Les vampires cessent de se déplacer.
Aucun vampire restant en vie	Buffy et les humains se déplacent aléatoirement d'une case à chaque tour.

Table 1 – Cas de tests et résultats attendus

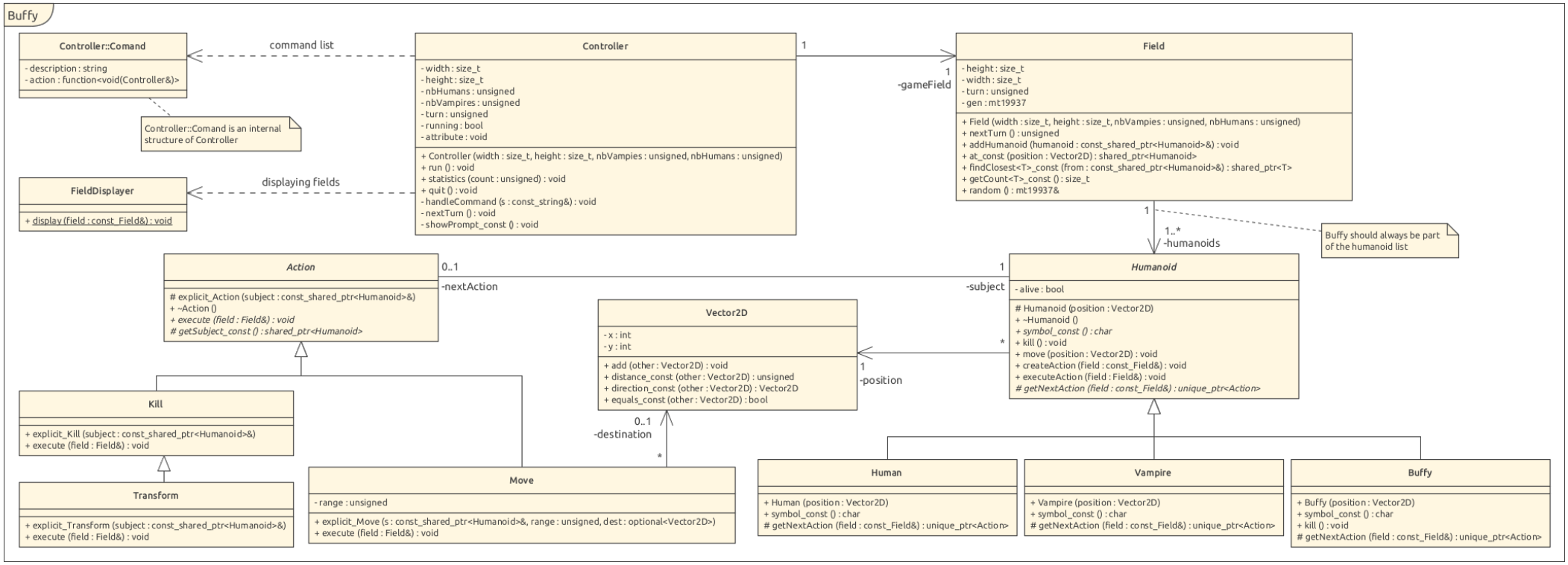

FIGURE 1 – Diagrammes de classe représentant l'ensemble de notre modélisation