|  |
| --- |
| Laboratoire 4 : Buffy |

|  |
| --- |
| Etudiants : Lange Yanik,  Tomic Mario  Professeur : Pier Donini  Assistant  : Grégoire Decorvet  09/06/2022 |

# Introduction

Le but de ce laboratoire est d’implémenter une simulation, où Buffy doit empêcher des vampires de tuer tous les humains.

Dans cette simulation, on entre une dimension du tableau de jeu, un nombre de humains et un nombre de vampires. Les humains se déplacent aléatoirement sur le tableau. Les vampires quant à eux pourchassent l’humain le plus proche et le tuent, avec une chance aléatoire de transformer l’humain en un vampire. Buffy se déplace vers le vampire le plus proche pour le tuer, avec une vitesse 2x supérieure.

Il est possible d'effectuer 10’000 simulations et établir un pourcentage de victoire. La victoire est décidée s’il n’y a plus de vampire sur le tableau, mais qu’il y a encore des humains. Il est également possible d’effectuer une simulation pas à pas.

# Résultats de la simulation

En effectuant 10’000 simulations sur une grille de dimension 50x50, contenant 10 vampires et 20 humains, on obtient un résultat se situant aux alentours de 46%.

# Choix d’implémentation

## Humanoïdes

Les humanoïdes sont modélisés au travers d’une classe abstraite. On retrouve sous-classes : Human, Buffy et les Vampires. Chacune de ces classes implémentent leurs propres comportements pour la fonction setAction.

Il aurait été possible de faire que Buffy soit une spécialisation de la classe Human. Un autre choix a cependant été fait. Pour la recherche d’un type spécifique d’humanoïdes, on utilise un dynamic\_cast afin de déterminer que l’humanoïde examiné est du type voulu.

Les vampires utilisent la méthode findNearestHuman dans le but de récupérer le prochain humain qu’ils vont essayer de tuer. Afin de ne pas retourner Buffy, il aurait fallu effectuer un second dynamic\_cast, dans le but de vérifier que l’humanoïde n’est pas Buffy. Finalement, Buffy dans son comportement n’est pas plus proche d’un humain que ce qu’elle est d’un vampire.

## Mouvement

Un mouvement est effectué au travers d’une décomposition de mouvement. Deux coordonnées (x, y) sont fournies, avec lesquelles un angle est calculé par rapport à l’humanoïde voulant de déplacer. On utilise cet angle ensuite pour effectuer une rotation du vecteur (1, 0). Ce vecteur prend alors une valeur arrondie pour x et y, faisant qu’il est possible de se déplacer en diagonale. On effectue cette action autant de fois que le nombre de cases que l’humanoïde peut se déplacer.

Cette méthode de déplacement est utilisée aussi bien pour pourchasser un humanoïde, que pour le déplacement aléatoire. Pour le déplacement aléatoire, il y a eu plusieurs itérations.

Initialement, il était question de choisir un point aléatoire sur le tableau. Cette méthode permettait de ne jamais avoir à générer des nombres aléatoires plus d’une fois par coordonnées, en plus de ne pas avoir à prendre en compte les limites du tableau. Cependant, cette méthode a été changée car celle-ci crée un biais assez conséquent. Les humanoïdes se retrouvaient à se diriger vers le centre en priorité, faisant que les résultats de la simulation étaient nettement moins bons (dans les alentours de 12%).

L’implémentation actuelle calcule un déplacement minimal et maximal pour chacune des coordonnées, prenant en compte les limites du tableau. Additionnellement, un biais a été ajouté dans le cas où les coordonnées choisies aléatoirement sont les mêmes que les coordonnées actuelles. Afin de forcer le mouvement, on décide que l’humanoïde se déplace d’une coordonnée x de –1, pour autant qu’il ne soit pas dans la coordonnée x = 0. Dans ce cas-là, le mouvement sera de +1.

## Kill et Transform

L’action Kill fonctionne de manière simple. On appelle le setter setAlive de l’humanoïde et on le met à false. On notifie ensuite le field que le bon type d’humanoïde a été tué. Transform est une spécialisation de Kill. En effet, la transformation n’est rien d’autre que l’action de tuer, avec une possibilité aléatoire de créer un nouveau vampire, là où l’humain a été tué.

## Smart Pointers

Le choix de l’utilisation des smart pointers a été fait pour une raison un peu particulière. Il fut un stade dans le labo où les classes n’étaient pas tout à fait définies. Lors de l’implémentation de findNearestHuman, le pointeur retourné ne produisait pas le résultat attendu. Le premier réflexe était alors de se référer au cours et y trouver les derniers concepts appris. C’est alors que la décision d’utiliser des smart pointers a été prise.

Au final, les problèmes ont été causés par des forward declaration qui n’ont pas été faites correctement. L’utilisation des smart pointers a été retenue malgré tout, afin d'entrainer l’utilisation de ceux-ci un peu plus en profondeur. C’est ainsi qu’on a pu constater que les smart pointers ne sont pas une solution magique à tout, et qu’il faut quand même les utiliser correctement, au risque de corrompre la heap.

Additionnellement, les actions prennent un weak\_ptr de humanoides, afin de ne pas empêcher la suppression de l’objet

# Tests effectués

* Les entrées de l’utilisateur sont prises en compte correctement
* Les humanoïdes qui se déplacent

# 

# Diagramme UML