16/06/2022

Rapport POA
Laboratoire 4 : Buffy

Valentin Kaelin et Alexandre Jaquier

Introduction

Dans le cadre de ce laboratoire, nous avons consolidé nos connaissances sur l'héritage en C++ et avons pour la première fois modélisé un projet assez conséquent en C++ (en dehors du projet PIN bien entendu). Nous avons dû implémenter une simulation dans la console dans laquelle Buffy essaie de sauver des humains se faisant tuer ou transformer par une horde de monstrueux vampires.

Choix de modélisation et d'implémentation

Hiérarchie des Humanoïdes

Pour les classes représentant les différents humanoïdes, nous avons un héritage assez simple composé tout d'abord une classe abstraite *Humanoid*. Nous aurions pu rajouté un niveau d'héritage en regroupant les chasseurs entre eux (Vampire et Buffy), mais leurs actions et conditions étant assez différentes, nous avons préféré rester sur quelque chose de simple.

Afin de pouvoir modifier les actions assignées aux *Humanoids*, nous avons créé une méthode abstraite et *protected getNextAction()*. Ceci nous permet de garder l'attribut de l'action en privé dans la classe *Humanoid* et de quand même pouvoir déclarer une action propre à chaque type d'*Humanoid*.

Nous avons une méthode *kill()* surchargée dans les *Humanoid* souhaités afin d'indiquer au Field que le nombre d'*Humanoid* d'un certain type a été modifié. Nous expliquerons plus en profondeur cet aspect par la suite avec les actions.

Constructeurs de copie

Dans les classes utilisant de l'allocation de mémoire dynamique, nous avons supprimé les constructeurs de copie ainsi que les opérateurs d'affectation afin de ne pas avoir à gérer la copie de ces ressources allouées. Nous avons choisi cette approche car la copie de ces classes ne fait pas vraiment de sens dans notre projet. Pour information, il s'agit des classes suivantes : *Humanoid*, *Field* et *Action*.

Actions

Nous avons créé une classe par type d'action héritant toutes de la classe abstraite *Action*. Ces actions fonctionnent légèrement comme des commandes du *Command Pattern*. Elles s'instancient à un instant T et peuvent être exécutées plus tard via leur méthode *execute*. C'est parfait pour notre utilisation car nous souhaitons assigner à chaque *Humanoid* une action avant d'en avoir exécuté la moindre.

Move

L'action de déplacement *Move* peut se réaliser de deux façons différentes :

- Soit un *Humanoid* cible est donné et dans ce cas le déplacement aura pour but de se rapprocher au maximum de la cible.
- Soit aucun cible n'est fournie, et le déplacement sera donc complètement aléatoire.

Pour les déplacements aléatoires, nous vérifions toutes les directions possibles en fonction de la position actuelle de l'*Humanoid* et de son nombre de cases parcourues et nous tirons aléatoirement une direction à additionner à la position actuelle. Ceci nous évite qu'un *Humanoid* ne bouge pas s'il est bloqué contre un mur et que son mouvement aléatoire lui demande de sortir du *Field* par exemple.

Dans le cas d'un déplacement vers une cible, nous calculons la direction optimale à appliquer à la position actuelle de l'*Humanoid* afin de se rapprocher le plus possible de sa cible. Nous calculons et additionnons cette direction à la position actuelle un nombre de fois égal au nombre de cases parcourues par tour pour l'*Humanoid*. Ceci nous permet par exemple d'atteindre une surface de 5x5 cases si l'*Humanoid* peut se déplacer de deux cases par tour.

Kill et Transform

Afin d'éviter qu'un *Humanoid* soit tué ou transformé plusieurs fois dans le même tour de la simulation, nous vérifions au préalable qu'il soit bien en vie. Lors de ses actions, c'est l'*Humanoid* qui va indiquer au *Field* qu'un nouvel humain ou vampire est mort (ou qu'un nouveau vampire est né lors du transformation). Cela nous facilite la vie car l'*Humanoid* qui meurt connait forcément son type et nous n'avons donc pas à le vérifier nous-même dans la boucle qui supprime les *Humanoid* morts par exemple.

Displayers

Nous avons réalisé un affichage des couleurs sur les différents OS (Unix/Windows). Pour ce faire, nous avons ajouté des accesseurs aux humanoïdes afin de récupérer leur couleur afin de ne pas avoir à créer des switchs pour associer un type d'humanoïde à une couleur.

Nous créons un vecteur à deux dimensions de pointeurs d'*Humanoid* représentant la grille à afficher. Nous faisons un tableau de pointeur et pas un tableau de string car l'affichage des couleurs sous Windows demande de faire un cout supplémentaire pour afficher la couleur, et pas uniquement un préfixe à ajouter comme sous Unix.

Les valeurs de ce vecteur sont réinitialisées dans la boucle de lecture afin d'éviter un parcours supplémentaire ou une suppression et recréation du vecteur.

Tous les affichages du programme ont été regroupés dans la classe Displayer afin de pouvoir facilement les modifier au besoin.

Field

Le *Field* contient la liste des *Humanoid* de la simulation. Nous avons mis à disposition des itérateurs constants afin d'itérer facilement sans pour autant compromettre l'encapsulation en interdisant la modification des *Humanoid*. Le *Field* contient deux compteurs : un pour le nombre d'humains vivants et un autre pour le nombre de vampires. Ces compteurs sont incrémentés / décrémentés lorsque lesdits humanoïdes sont tués / transformés. Grâce à ces compteurs, nous pouvons savoir très facilement si la partie est en cours, gagnée ou perdue sans avoir à parcourir tous les *Humanoid* du *Field*.

De plus, nous avons une méthode générique dans le *Field* nous permettant de trouver l'*Humanoid* du type souhaité le plus proche. Nous utilisons une *dynamic_cast* afin d'accepter également de potentielles sous-classes de la cible dans un soucis d'évolutivité.

Controller

Le *Controller* contient la logique principale du programme. C'est à lui de réagir aux interactions de l'utilisateur. Il délègue l'affichage au *Displayer*.

Random

Pour les valeurs aléatoires de ce laboratoire, nous avons choisi d'utiliser la méthode plus actuelle de génération en C++ grâce à une distribution uniforme de la classe *uniform_int_distribution*. Cette façon de faire est actuellement plus recommandée que l'ancienne fonction r*and()*.

Réponse à la question

Pourcentage de succès de Buffy pour les conditions initiales : grille de 50x50, 10 vampires et 20 humains.

Nous avons un pourcentage de succès d'environ **54.51%**. Ce résultat varie légèrement d'un calcul de statistiques à l'autre mais pas de manière significative (ouf !).

Tests effectués

Concernant les tests de ce laboratoire, nous nous sommes surtout contentés de faire des tests des fonctionnalités globales du programme et pas des sortes de tests unitaires de chaque classe créée. Nous partons du principe que ces tests plus généraux testent également le bon fonctionnement des différentes structures de données créées.

Test	**Résultat attendu et observé**
Lancement du programme avec une taille fournie, un nombre d'humains et un nombre de vampires	Le Field apparaît avec nombre d'humains et de vampires souhaités ainsi qu'une Buffy. Tous ces humanoïdes ont des positions aléatoires.
L'utilisateur lance le programme avec des arguments invalides (ex : manquants, valeurs < 0, valeurs non numériques)	Un message d'erreur apparaît et le programme s'arrête.
L'utilisateur lance le programme avec des arguments valides mais une ou plusieurs valeurs vaut 0.	Un message d'erreur apparaît et le programme s'arrête.
L'utilisateur entre l'input « n »	L'affichage se met à jour, chaque humanoïde de la simulation a reçu une action et l'a effectuée.
L'utilisateur appuie sur sa touche entrée	L'affichage se met à jour, chaque humanoïde de la simulation a reçu une action et l'a effectuée.
L'utilisateur entre l'input « q »	Le programme se ferme sans soucis, toutes les allocations dynamiques sont bien désallouées.
L'utilisateur entre l'input « s »	Un nombre de simulation fixé (10'000) est démarré avec les paramètres rentrés au lancement de l'application. Une fois terminées, le pourcentage de victoire de Buffy est affiché.
L'humanoïde humain passe un tour de simulation.	L'Human se déplace d'une case aléatoirement et reste à l'intérieur du terrain de jeu.
Un vampire se trouve à plus d'une case de l'humain le plus proche.	Le vampire se rapproche de l'humain le plus proche d'une case.

Une vampire se trouve à une case ou moins d'un humain.	Le vampire attaque et transforme l'humain en vampire ou le tue sans bouger. Si l'humain est transformé en vampire, il ne bouge également pas.
Buffy se trouve à plus d'une case du vampire le plus proche.	Buffy se rapproche du vampire le plus proche de deux cases par tour.
Buffy se trouve à moins d'une case du vampire le plus proche.	Buffy tue le vampire sans bouger.
Plus aucun humain n'est vivant	Les vampires de la simulation ne bougent plus aux différents tours.
Plus aucun vampire n'est vivant	Buffy et les humains se déplacent aléatoirement d'une case par tour.

Le programme a également été lancé avec Valgrind afin de pouvoir détecter d'éventuelles fuites de mémoire.