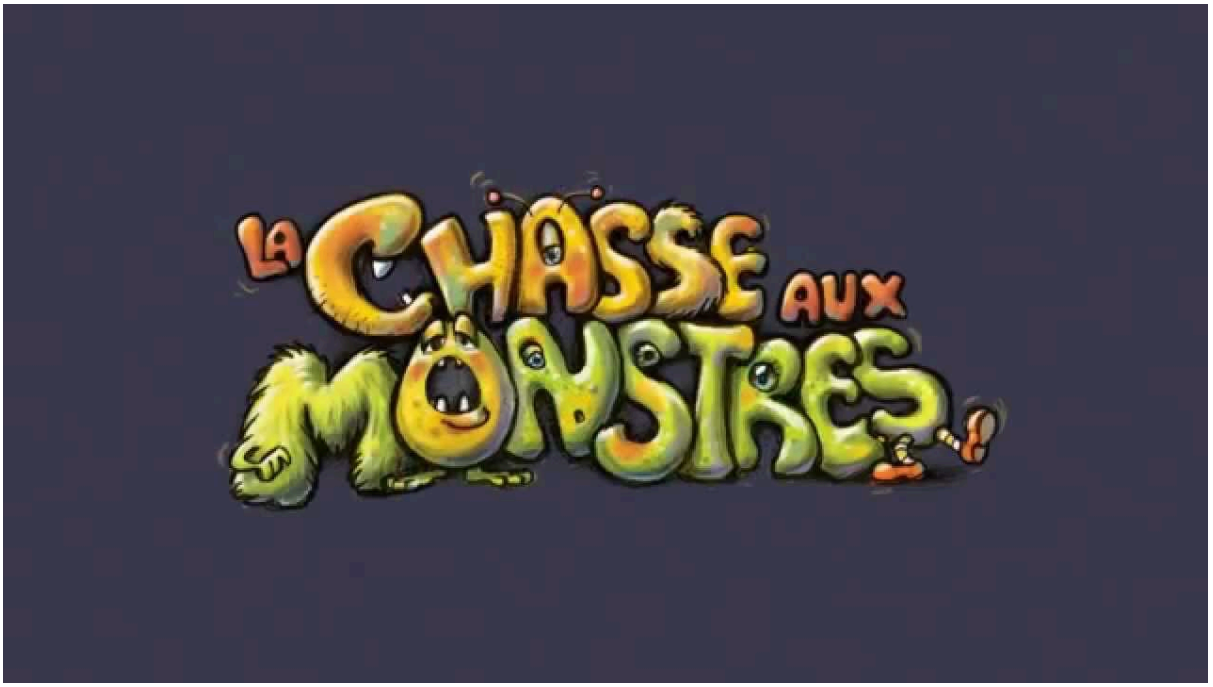
Rapport SAÉ3.02 Dev. applications
Chasse au monstre



Groupe J2 | Top Jessy | Franos Théo | Fourmaintraux Camille | Debacq Arthur

Génération de labyrinthe

Description:

L’algorithme de génération de labyrinthe est dans la classe MazeGenerator du paquetage Management.
Nous avons mis en œuvre un algorithme pour générer des labyrinthes parfaits dans notre programme. Cet algorithme fonctionne en partant d'un labyrinthe vide et en ajoutant progressivement des murs. Il offre une grande flexibilité en permettant de modifier sans problèmes la taille du labyrinthe en hauteur et en largeur. De plus, la possibilité de choisir un pourcentage d'apparition des murs ajoute une dimension de personnalisation. Nous utilisons deux tableaux de booléens pour la génération des murs, l'un avec une génération de murs à 100% qui est utilisé par l’algorithme pour éviter des erreurs, et l'autre qui copie ce premier tableau, mais en appliquant le pourcentage de risque d'apparition des murs pour le jeu. C’est ce deuxième labyrinthe qui est effectivement utilisé pour le jeu.

Modularité et Qualité:

Notre algorithme de génération de labyrinthes se distingue par sa modularité et sa capacité à s'adapter à différentes tailles et paramètres. La qualité des labyrinthes générés est soulignée, soit par leur caractère parfait lorsque l’on choisit de générer un labyrinthe avec 100% des murs, soit dans le cas contraire souligné par une très bonne répartition des murs parmi le plateau de jeu, ce qui garantit une expérience de jeu optimale dans tous les cas.

L’IA

Pour les IAs nous avons 3 niveaux de difficultés: **facile, intermédiaire** et **difficile**. Nous avons développé nos propres algorithmes pour les IAs du chasseur et de l'IA moyenne du monstre.

Pour les IAs faciles on utilise simplement un random.

-Pour **l’IA facile du chasseur**, l'algorithme choisit une case aléatoirement du labyrinthe et tire dessus.

-Pour **l’IA facile du monstre** l’algorithme choisit aléatoirement une des 8 cases autour de lui et ensuite vérifie si le monstre peut se déplacer à cette case (vérifie s' il y a un mur), si ce n’est pas le cas le monstre se déplace.

-Pour **l'IA intermédiaire du chasseur**, il tire encore aléatoirement sauf qu'à la différence du mode facile il prend en compte de si c'est un mur et l'enregistre afin de ne plus jamais tirer dessus.

-Pour **l'IA intermédiaire du Monstre**, il se déplace vers la sortie en comparant sa position avec celle de la sortie. C'est-à-dire si sa position x est plus basse que la position x de la sortie, alors on incrémente x que si il peut pas se déplacer en x car il peut y avoir un mur, puis on fait de même pour y et si c'est encore impossible, on se déplace aléatoirement.

-Pour **l'IA difficile du chasseur**, elle est identique à celle de l'IA intermédiaire mais à la différence que, si le chasseur tire sur une case avec une trace du monstre (une trace correspondant au numéro de tour où le montre est passé sur une case) il va tirer à côté de cette trace, si il tire sur une autre trace qui est plus récente, il va tirer à côté de cette autre trace, ce qui fait que petit à petit il remonte jusqu'au monstre.

-Pour **l'IA difficile du monstre**, nous avons opté pour l'algorithme A*. C'est un algorithme de recherche de chemin largement utilisé pour résoudre des problèmes de chemin dans des graphes, comme la recherche du chemin le plus court dans un environnement comme ici, les labyrinthes. L'algorithme A* garantit de trouver le chemin optimal, c'est-à-dire le chemin avec le coût total le plus bas. L'algorithme reconstruit le chemin optimal en remontant à partir du nœud d'arrivée jusqu'au nœud de départ en suivant les liens entre les nœuds. Cet algorithme est dans la classe AStar du paquetage strategy/monster/astar. Il utilise la classe Node qui est dans le même paquetage et est utilisé par IAhardcoreMonster dans le paquetage strategy/monster/astar.

Structure de données

Pour ce projet nous utilisons de nombreuses structures de données, afin de communiquer plus efficacement les informations entre les classes.
Par exemple :

- Les classes “saves et data” : SaveMonsterData, SaveHunterData, SaveExitData, SaveMazeData, SaveManagementData, ces classes contiennent toutes les données absolument nécessaires au déroulement du jeu et sont donc utilisées pour la sauvegarde d’une partie (représenté par la classe Save).
  GameplayMonsterData/GameplayHunterData contiennent les données sur les règles de la partie concernant respectivement le monstre et le chasseur. (De combien de case le monstre peut-il se déplacer, si sa vision est limitée, ...)
  Les noms sont assez explicites quant à quel classe ces structures sont utilisées, GameplayMonsterData et SaveMonsterData sont utilisés dans la classe Monster, de même pour Hunter, SaveManagementData dans la classe Management, SaveMazeData dans la classe Maze, etc...
  Ces structures de données sont de loin les plus essentielles car elles permettent de communiquer efficacement les données entre les classes, notamment en jeu et à la sauvegarde. Ces classes sont donc serializable et utilisent que des objets qui le sont aussi.

- La classe Maze, qui est la représentation d’une partie. Cette classe contient Monster et Hunter ainsi que toutes les méthodes pour jouer au jeu.

- Monster / Hunter, qui sont des classes contenant toutes les informations sur chacun des joueurs. Ce sont ces classes et leurs stratégies associées qui déterminent les actions et prochains déplacements des joueurs, mais c’est la classe Maze qui décide de si oui ou non à le droit de se déplacer là où le joueur le veut. Monster et Hunter sont isolés et ne peuvent interagir avec aucunes autres classes, seul Maze peut interagir avec eux et changer leur contenu. Maze est utilisé dans la classe Management.

- La classe Management gère les menus et le déroulement d'une partie. Cette classe contient les Views et le Maze. C'est la classe essentielle de notre programme.
  Elle est utilisé dans la classe de démarrage de notre programme : MonsterHunt.java

- Les classes "cells" qui permettent l'affichage du jeu. Elles sont utilisées dans les classes view MonterView et HunterView. Sans elles, afficher le jeu serait beaucoup plus difficile, car elles permettent de regrouper tout ce qui permet l'affichage. Notre jeu affiche non pas avec un Canvas ou un contexte graphique puis des dessins mais avec des rectangles dans des Groupes. Cell étend la classe rectangle de JavaFX afin de pouvoir leur ajouter ce dont nous avions besoin pour le projet. CellWithText étend Cell et ajoute simplement la possibilité d'ajouter du texte à la Cellule.

- La classe Theme, qui gère la partie graphique de l'affichage (les couleurs, les images, etc...). Ces structures diversifient l'apparence du jeu et le rendent plus attrayant. Cette structure de données est utilisée dans la classe Management.

- La classe DisplayValues qui contient les données de la fenêtre hauteur, largeur, position et autres, sans cette classe, la fenêtre reviendrais constamment à sa taille ou position originel à chaque interaction ce qui est embêtant. Cette structure de données est utilisée dans la classe Management.

Efficacité des Algorithmes

Justifications des choix:

Nos choix algorithmiques et de structures de données sont efficaces en termes d'opérations et de rapidité d'exécution. Notre algorithme de génération de labyrinthe est assez efficace, il peut produire des immenses labyrinthes parfait en un temps négligeable. L'algorithme A* pour l'IA difficile du monstre offre des bénéfices significatifs. A* utilise une heuristique pour guider la recherche du chemin optimal, ce qui le rend plus performant que d'autres algorithmes tels que Dijkstra dans le contexte de ce jeu. En utilisant la combinaison de la fonction de coût réel jusqu'à un nœud et de l'estimation heuristique, A* évite d'explorer inutilement des chemins coûteux. Il assure une exploration plus ciblée et une meilleure efficacité, contribuant ainsi à une expérience de jeu plus fluide.

Axes d'améliorations:

En nous concentrant sur l'IA du monstre, nous avons laissé énormément de caractéristiques améliorables chez l'IA du chasseur. Nos IAs ne sont pas parfaites et pourraient êtres améliorés, notamment en essayant de mieux optimiser l'algorithme a* de l'IA du monstre et en rendant l'IA du chasseur plus réactive.