SQUAD AI

LAHALLE François et PIRIS Marius
GP3

Description du projet

Exercice décisionnel et déplacements en IA.

Objectifs:
Réaliser une IA de NPCs se déplaçant en escouade et pouvant interagir avec le joueur, dans le cadre d'un jeu d'action temps réel.

Dans le cadre de notre projet en Intelligence Artificielle, nous avons travaillé sur la conception d'une IA appliquée à des NPCs dans un jeu d'action en temps réel. Ce projet nous a permis de mettre en pratique les notions vues en cours, notamment en matière de déplacement et de prise de décision pour des agents autonomes.

Nous avons développé un système où des NPCs alliés interagissent de manière autonome avec le joueur, en adoptant plusieurs comportements : suivre le joueur, tirer en soutien, le protéger face à des attaques ennemies, ou encore le soigner lorsque sa santé est basse. Chaque NPC a un rôle particulier, mais peut s'adapter à la situation en fonction des besoins du joueur. Ces NPCs se déplacent également en formation autour du joueur, offrant ainsi un soutien cohérent et structuré.

Le projet a été réalisé sur Unity 2022.3.47f1 en utilisant les outils natifs du moteur, notamment les NavMesh Agents pour la gestion des déplacements.

Conception

Flocking

Pour gérer le comportement de groupe (flocking) de nos NPCs alliés, nous avons conçu un système basé sur plusieurs préfabriqués de formations. Chaque préfabriqué représente une formation spécifique que les NPCs peuvent adopter en fonction de la situation de jeu (carré, cercle, en “V”...). Le prefab central "Flock" est chargé de coordonner le déplacement de l’escouade par rapport à la position du joueur et d'attribuer les bonnes positions aux NPCs en fonction de la formation choisie.

Le prefab "Flock" joue un rôle clé dans la dynamique de groupe : il contient différents modules dédiés à la gestion des rôles des NPCs et à la prise de décision. Ces modules calculent quels agents doivent être envoyés pour exécuter certaines actions spécifiques.

Par exemple, lorsqu’un joueur est gravement blessé, un pourcentage des NPCs médecins est assigné pour aller soigner le joueur, mais pas la totalité, afin de maintenir un équilibre dans les autres fonctions du groupe. De la même manière, une portion des NPCs protecteurs (guardians) est déployée pour protéger le joueur contre les ennemis, tandis qu’un autre groupe de NPCs est assigné au tir de couverture, permettant de répondre à plusieurs menaces à la fois.

Enfin nous avons 4 scripts, un pour chaque classe, qui seront affectés aux agents en eux même et qui auront des fonctions permettant d’effectuer leurs actions comme soigner et protéger et ces fonctions seront appelés par les différents nœuds d’action.

Behavior Tree

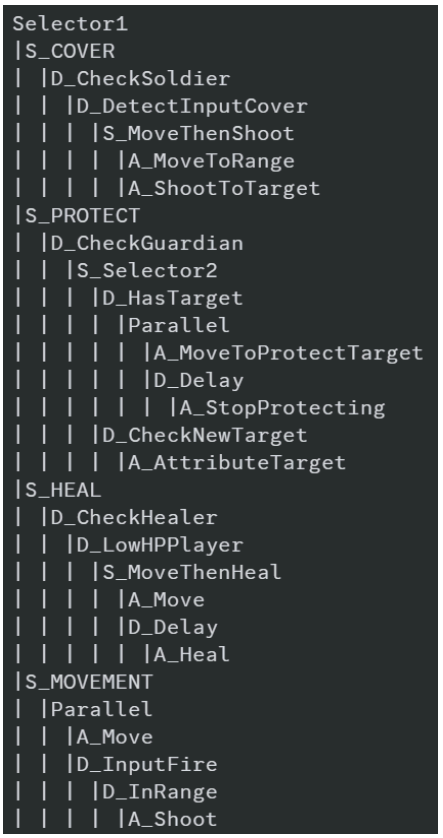
Notre Behavior Tree se compose de différents types de nœuds, qui découlent tous d’une classe abstraite "Node". Cette architecture permet de définir un comportement de manière standardisée et d'assurer la cohésion entre les différentes parties du système.

- Nœuds composites : Ce sont des nœuds qui contiennent d'autres nœuds. Ils permettent de structurer le comportement en sous-tâches et de déterminer l’ordre ou la priorité dans laquelle ces sous-tâches seront exécutées. Les nœuds composites incluent des structures telles que les selectors, les séquences et les parallèles (le node parallèle permet de lancer les nœuds suivants même s’il y a un état en cours, toutefois s’il y a un état d’échec alors il s’arrête).

- Nœuds d'action : Ce sont les feuilles du Behavior Tree, c’est-à-dire les nœuds qui réalisent les actions concrètes dans le jeu, comme suivre le joueur, tirer sur un ennemi, ou se soigner. Ils exécutent directement des instructions précises et renvoient le résultat de l’action (succès, échec ou en cours) à leur nœud parent.

- Nœuds de décoration : Ces nœuds servent à modifier ou à conditionner l'exécution des nœuds enfants. Par exemple, ils peuvent empêcher un nœud de s'exécuter si certaines conditions ne sont pas remplies, ou encore répéter une action jusqu'à son succès.

Voici le Behavior Tree que nous avons conçu, il est constitué d'un nœud "racine" (un selector) duquel découle 4 séquences d'actions : feu de couverture, protection du joueur, soin du joueur et mouvement par défaut. L'arbre est parcouru de haut en bas toutes les x secondes. Dès qu'une séquence renvoie un état d'échec, on passe à la suivante. Les nodes commençant par "A" sont des actions, par "D" sont des décorateurs et par "S" sont des séquences.



Voici une brève description des différentes node utilisées :

- Selector exécute ses noeuds enfants et lorsque l'un d'entre eux renvoie "échec", ce dernier essaie d'exécuter l'enfant suivant et ainsi de suite.
- Séquence fait la même chose sauf que pour passer à l'exécution de l'enfant suivant, il faut que l'enfant précédent renvoie un succès.
- D_Check… Vérifie la classe de l'allié, check guardian continue si l'agent est de la classe "guardian", sinon cela renvoie échec.
- A_MoveTo… Déplace le joueur jusqu'à une cible définie. Par exemple, "A_MoveTo_Protect_Target" déplace l'agent jusqu'au point enregistré (dans un blackboard) pour se placer entre le joueur et les tirs entrant.
  Si on atteint le point enregistré, on retourne un succès, sinon, on retourne "en cours".
- D_Delay attend quelques secondes avant d'exécuter la suite.
- Les autres nœuds sont plus spécifiques.

Réalisation

Flocking

Le fonctionnement du flocking repose sur des formations définies, et le système peut attribuer des rôles spécifiques aux NPCs, en envoyant certains agents soigner, protéger ou couvrir le joueur (des logos spécifiques sont affichés à côté des barres de vie des NPCs).

Classe Flock

Le prefab "Flock" contient une liste d'agents appelés flockAgents, qui représente les NPCs formant le groupe. Au démarrage, ce prefab initialise les NPCs en leur attribuant des

positions calculées par une formation sélectionnée, comme la formation en V, en carré ou en cercle. Ces formations sont représentées par des prefabs de formation (Formation), et chaque formation dispose d'un algorithme pour placer les agents autour du joueur avec des espacements définis par distanceBetweenAgent.

Gestion des rôles dans le flocking

Chaque agent du flock a un rôle particulier, et il est possible d'attribuer différents types d'agents en fonction des besoins du joueur. Ces rôles sont assignés via le prefab roleAgentPrefab, qui contient plusieurs types d'agents avec des comportements distincts (soigneurs, protecteurs, etc.). Par exemple :

- Healers : Une fraction des NPCs soignants est envoyée pour soigner le joueur lorsqu'il est blessé, mais pas tous les agents du même rôle.
- Protecteurs : De la même façon, un pourcentage des agents protecteurs est envoyé protéger le joueur lors d'une attaque.
- Tireurs de couverture : Certains agents sont désignés pour fournir un tir de couverture selon la situation, sans que l'ensemble des agents du même rôle n'y soit affecté.
- Cette gestion est rendue possible par le module ProtectGroup pour les protecteurs et HealingPlayerGroup pour les soigneurs. Chaque groupe est capable de déterminer le nombre d'agents nécessaires pour une tâche spécifique, afin de maintenir un équilibre dans l'escouade.

Calcul des positions dans la formation

Lorsqu'une formation est appliquée (par exemple la formation en carré avec ApplySquareFormation()), le système recalcule la position de chaque agent par rapport au joueur, en fonction du nombre d'agents et de la distance entre eux. Les positions sont ensuite réattribuées à chaque agent avec la méthode Recalculate(), qui met à jour les cibles de déplacement des NPCs.

Sélection des agents les plus proches

La méthode GetCloserAgents() permet de sélectionner un certain pourcentage d'agents qui sont les plus proches d'une cible (par exemple, un endroit où le joueur est blessé ou attaqué). Cela garantit que seuls les agents les plus appropriés (en termes de distance) sont affectés à une tâche, limitant ainsi les déplacements inutiles et optimisant la réactivité de l'escouade.

Classe FlockAgent

Chaque FlockAgent représente un NPC dans le groupe. Les agents sont capables de se déplacer vers une cible spécifique grâce à leur composant AIAgent. Ils possèdent plusieurs propriétés :

- Offset : représente l'écart entre la position du leader (le joueur) et la position de l’agent dans la formation.
- Move() : déplace l’agent vers la position cible calculée en tenant compte de l’offset par rapport à la position du leader.
- RecalculatePosition() : recalcule l'offset de l'agent lorsque le joueur change de position ou de direction.

Les agents peuvent aussi remplir différents rôles, comme le soin avec HealingPlayer, la protection avec ProtectPlayer, ou le tir de couverture avec CoverFire.

Formations

Les formations, comme la SquareFormation, calculent les positions des NPCs autour du joueur. Chaque formation a ses propres paramètres, comme le nombre de lignes et de colonnes pour la formation en carré. Les positions sont calculées de façon à ce que les NPCs soient uniformément répartis autour du joueur, tout en tenant compte de la taille de la formation et du nombre d’agents.

Behavior Tree

Classe Node

Node est la classe de base abstraite dont héritent tous les nœuds du Behavior Tree. Elle contient une énumération State, qui définit trois états possibles pour chaque nœud : Running, Success et Failure.

Méthodes principales :
Start() : Cette méthode sert à initialiser chaque nœud. Elle cherche le composant BehaviorTreeRunner à travers la fonction FindParentWithTag(), qui remonte la hiérarchie des objets pour trouver un parent avec le tag "Tree". Cela permet de connecter chaque nœud au Behavior Tree global.

UpdateNode() : C’est la méthode qui met à jour l’état du nœud. Si le nœud n’a pas encore démarré, OnStart() est appelé pour initialiser les tâches. Ensuite, le nœud exécute OnUpdate(), qui met à jour son état. Si l’état devient Success ou Failure, OnStop() est exécuté et le nœud est réinitialisé pour un prochain cycle.

Les méthodes OnStart(), OnStop(), et OnUpdate() sont abstraites et sont définies dans chaque sous-classe pour gérer des comportements spécifiques.

Classe CompositeNode

Spécificités :
children : Ce champ est une liste qui contient les enfants du nœud composite. Ce sont ces nœuds qui seront exécutés les uns après les autres ou selon des règles spécifiques (comme des selectors ou des sequences).

Start() : Lors de l'initialisation, cette méthode récupère tous les enfants du nœud dans la hiérarchie des objets et les ajoute à la liste children. Cela permet de construire une structure hiérarchique dans laquelle chaque nœud peut exécuter des sous-nœuds de manière organisée.

Classe ActionNode

Spécificités :
npc : Ce champ stocke une référence vers le NPC auquel le nœud d'action est associé. Cela permet de manipuler directement le NPC lorsqu'une action doit être réalisée.

Start() : Comme pour les autres nœuds, cette méthode récupère la référence du NPC parent via FindParentWithTag() en cherchant le tag "NPC". Chaque nœud d'action peut ainsi s'exécuter en fonction du contexte.

Classe DecoratorNode

Spécificités :
child : Ce champ contient le nœud enfant supervisé par le décorateur. Celui-ci est initialisé lors du Start() en récupérant le premier enfant de la hiérarchie.

Classe BehaviorTreeRunner

Variables principales :
BlackBoard Data : C'est un élément central du système. Le BlackBoard est un espace de stockage partagé qui contient toutes les données nécessaires pour les décisions de l'IA. Les différents nœuds peuvent y accéder pour consulter et modifier des informations, comme la position du joueur ou l'état de la santé d'un NPC.

rootNode : Ce champ fait référence au nœud racine du Behavior Tree, qui est le point de départ de toute la logique de l'arbre. Ce nœud est assigné dynamiquement lors du lancement du jeu.

treeState : Cette variable stocke l'état actuel du Behavior Tree (Running, Success, Failure). Elle permet de suivre si l'arbre est en cours d'exécution ou s'il a terminé son travail avec succès ou non.

timerUpdate et currentTimer : Ces variables contrôlent le temps entre chaque mise à jour de l'arbre. Plutôt que d'appeler la fonction d'update à chaque frame, on introduit un intervalle pour limiter la fréquence des mises à jour, optimisant ainsi les performances de l'IA.

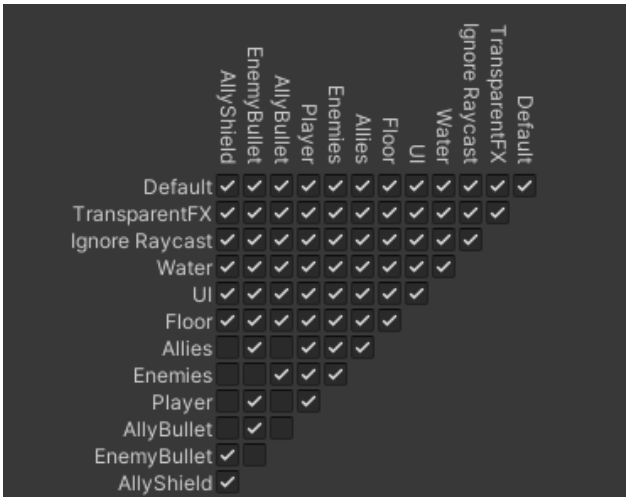
Méthodes principales :
Start() : Cette méthode est appelée une fois lorsque le jeu commence. Elle cherche à identifier le nœud racine (rootNode) en cherchant le premier enfant qui possède un composant de type Node. Une fois trouvé, ce nœud devient le point de départ de l'arbre.

Update() : Cette méthode est exécutée à chaque frame. Elle vérifie d'abord si un nœud racine est présent, en utilisant la variable hasRootNode. Si le nœud racine existe et que l'état de l'arbre est toujours Running, l'update est effectuée à intervalles réguliers (basé sur timerUpdate). Lorsque le temps défini par timerUpdate est écoulé, la méthode UpdateNode() du nœud racine est appelée pour mettre à jour l'état de l'arbre.

Si aucun nœud racine n'est trouvé, l'état de l'arbre passe à Failure, indiquant que quelque chose n'a pas fonctionné dans la configuration du Behavior Tree.

Documentation

Lien du itch.io (jouable sur navigateur) : ici
Scène à lancer : “Menu”
Scène de jeu principale : “GameScene”
Dossier des scènes : Assets/Scenes
Paramètres de physique du projet :



Tutoriel

Touches :
Déplacement : Z,Q,S et D
Tir : clique gauche
Tir de couverture : clique droit (Activer/Désactiver)
Changer de formation : C (Circle Formation) ou V (V Formation) ou B (Square Formation)

Le Guardian vient se mettre devant le joueur quand ce dernier se fait attaquer.
Le Healer vient soigner le joueur quand sa vie est basse, il faut que le joueur reste immobile.
Détruire les tourelles augmente le score.

Une vidéo de gameplay est disponible sur le git.