**Présentation générale du projet**

Le but de notre travail était d’implémenter un framework Unity permettant la gestion du comportement d’une entité du jeu à travers un arbre de comportement.

Notre arbre devait pour ce faire inclure les opérateurs :

* Sélecteurs
* Séquence
* Répéteur
* Etc…

**Techniques utilisées**

Afin de pouvoir mener à bien notre travail, nous avons jugé bon de créer notre propre librairie de gestion d’arbre.

Dans cette optique nous avons pris connaissance des différentes méthodes d’implémentation d’un arbre qui existent déjà afin de savoir comment nous devrions nous y prendre.

Nous avons également dû implémenter une partie de l’algorithme de déplacement sur un chemin car Unity ne nous nous proposait pas tout ce dont nous avions besoin.

Dans la suite, il sera présenté les différentes techniques utilisées et leurs impacts sur le jeu.

1. **Arbre de comportement (Behavior Tree)**

Un **arbre de** comportement est un outil d'aide à la [décision](https://fr.wikipedia.org/wiki/D%C3%A9cision) représentant un ensemble de choix sous la forme [graphique](https://fr.wikipedia.org/wiki/Repr%C3%A9sentation_graphique) d'un [arbre](https://fr.wikipedia.org/wiki/Arbre_(graphe)). Les différentes décisions possibles sont situées aux extrémités des branches (les « feuilles » de l'arbre), et sont atteints en fonction de décisions prises à chaque étape. L'arbre de décision est un outil utilisé dans des domaines variés tels que la [sécurité](https://fr.wikipedia.org/wiki/S%C3%A9curit%C3%A9), la [fouille de données](https://fr.wikipedia.org/wiki/Fouille_de_donn%C3%A9es), la [médecine](https://fr.wikipedia.org/wiki/M%C3%A9decine), etc. Il a l'avantage d'être lisible et rapide à exécuter. Il s'agit de plus d'une représentation calculable automatiquement par des algorithmes d'apprentissage supervisé.

Un aspect fondamental des arbres de comportement est que, contrairement à une méthode d’une base de code, un nœud ou une branche particulière de l’arbre peut prendre plusieurs tics du jeu à compléter. Dans l’implémentation de base des arbres de comportement, le système parcourt chaque image depuis la racine de l’arbre, testant chaque nœud dans l’arbre pour voir ce qui est actif, en revérifiant tous les nœuds le long du chemin, jusqu’à atteindre le nœud actuellement actif coché à nouveau.

Un arbre de comportement est composé de plusieurs types de nœuds, mais certaines fonctionnalités principales sont communes à tout type de nœud dans un arbre de comportement. C'est qu'ils peuvent renvoyer l'un des trois statuts suivants :

* *Succès*
* *Échec*
* *Fonctionnement*

Les deux premiers, comme leur nom l'indique, informent leur parent que leur opération a été un succès ou un échec. La troisième signifie que le succès ou l'échec n'est pas encore déterminé et que le nœud est toujours en cours d'exécution. Le nœud sera à nouveau coché la prochaine fois que l'arborescence est cochée, après quoi il aura à nouveau la possibilité de réussir, d'échouer ou de continuer à s'exécuter.

Cette fonctionnalité est essentielle à la puissance des arbres de comportement, car elle permet au traitement d'un nœud de persister pendant de nombreuses périodes du jeu. Par exemple, un nœud Promenade offrira le statut En cours d'exécution pendant le temps où il tente de calculer un chemin, ainsi que le temps nécessaire au personnage pour se rendre à l'emplacement spécifié. Si la recherche de chemin a échoué pour une raison quelconque, ou si une autre complication est survenue pendant la marche pour empêcher le personnage d'atteindre l'emplacement cible, le nœud renvoie l'échec au parent. Si, à un moment quelconque, l'emplacement actuel du personnage est égal à l'emplacement cible, il renvoie succès indiquant que la commande Walk a été exécutée avec succès.

Cela signifie que ce nœud a, de manière isolée, un contrat de fonte défini pour la réussite et l'échec, et tout arbre utilisant ce nœud peut être assuré du résultat reçu de ce nœud. Ces statuts se propagent ensuite et définissent le flux de l’arborescence, afin de fournir une séquence d’événements et différents chemins d’exécution le long de l’arborescence afin de s’assurer que l’intelligence artificielle se comporte comme souhaité.

1. **Nœud**

Les nœuds d'action exécutent une seule commande, mettent à jour l'état dans le contexte et renvoient le statut de l'arborescence. La commande est définie comme une fonction qui prend une coche de jeu et le contexte.

Il existe plusieurs types de nœuds à savoir :

* **Séquence**

Une séquence visitera chaque noeud enfant dans l'ordre. Si un nœud enfant échoue, il renverra immédiatement l'échec au nœud parent. Si le dernier nœud enfant de la séquence réussit, la séquence renvoie succès à son nœud parent.

* **Sélecteur**

Un sélecteur visite les nœuds enfants en séquence jusqu'à ce qu'il en trouve un qui réussit. Pour les nœuds enfants qui échouent, il est déplacé vers le nœud enfant suivant. Retourne un échec si aucun des nœuds enfants ne réussit.

* **Nœud action**

Les nœuds d'action exécutent une seule commande, mettent à jour l'état dans le contexte et renvoient le statut de l'arborescence. La commande est définie comme une fonction qui prend une coche de jeu et le contexte.

1. **Contexte**

L'objectif de la bibliothèque étant de faciliter la réutilisation de parties d'arborescence, il n'est pas recommandé d'accéder aux variables globales à partir des nœuds. Au lieu de cela, l'objet de contexte défini par l'utilisateur est transmis à tous les nœuds. Le nœud doit lire le dernier état de l'objet contexte et mettre à jour le contexte en conséquence. Le contexte de données dépend fortement du cas, il n'y a donc pas de limite. Ce sera juste une classe à définir.

1. **Game Tick**

La mise à jour de notre arbre se faisant à chaque frame ; il est important de passer en paramètre un temps de frame ou de jeu qui s’incrémente à chaque appel de l’arbre.

* **Implémentation**
* **Création d’un arbre**

TypeDeNœud arbre = new TypeDeNœud<Contexte>(List de Nœuds action)

* **Séquence**

new SequenceNode<Context>(node1, node2, node3, …);

* **Sélecteur**

new SelectorNode<Context>(choice1, choice2, choice3);

* **Contexte**

public class MyContext

{

public String str;

public GameObject obj;

}

* **Action**

var action = new ActionNode<Context>((timeTick, ctx) =>

{

return TreeStatus;

});

* Répéteur

La répétition se fait en déclarant un nœud action unique dans une séquence qui retourne un Status.Running.

* **Logique**

Un agent commence avant tout à vérifier s’il peut attaquer le joueur. Si oui, il le fait, sinon il patrouille. Mais si pendant sa patrouille il repère le joueur, il l’attaque aussitôt où après un moment de préparation. Les agents disposent également de trois états (Patrouille où ils sont en noir, Suspicion où ils sont en cyan et Attaque où ils sont en rouge). Le temps d’attaque d’un agent varie également selon la situation et parfois un agent peut abandonner la poursuite s’il trouve qu’il ne peut pas rattraper le joueur.

En cas de détection, l’agent ira jusqu’à la dernière position connue du joueur afin d’essayer de le repérer. Si c’est la cas il le poursuit de nouveau, sinon il continue sa patrouille., il le poursuit de nouveau

Il peut aussi communiquer avec les autres agents autour afin de savoir vers où aller pour bloquer le joueur assez vite.

Cela rend le jeu à la fois complexe vu la logique floue intégrée et en même temps pas frustrant vu que le joueur peu s’en sortir avec un peu de réflexion.

En cas de mort, le joueur est spawné à une nouvelle position aléatoire sur la map.

* **Framework**

Notre framework est basée sur une bibliothèque que nous avons écrite nous même et contient donc encore sûrement des erreurs.

1. **Navigation Mesh**

Unity intègre déjà un système de navigation « Navesh ». Il nous permet de directement appeler l’algorithme A\* qui nous permet de rechercher le chemin le plus court vers un emplacement. Nous économisons ainsi du temps et de la mémoire avec l’usage de A\* qui est un algorithme réputé pour sa rapidité et son efficacité dans le domaine du jeu.

Cependant, cet outil ne remplissait pas toutes les fonctionnalités dont nous avions eu besoin. Nous avons donc repris une bibliothèque un peu améliorée de NavMesh basée sur ce dernier. Nous avons juste améliorer ce dernier afin de pouvoir directement définir le type des objets pendant l’exécution par exemple. On a donc défini plusieurs autre type comme les NavMesh Obstacle, les NavMesh Surface, les NavMesh Modifier, les NavMesh Link, etc…Avec ces nouveaux types, nous avons rendu la navigation plus dynamique et plus fluide.

La combinaison de notre arbre et de la navigation nous a donc permis de créer des agents intelligents capable de prendre des décision seuls ou avec ses pairs. Il peut également se frayer un chemin optimal vers une position donnée du monde.

Nous avons opté pour une génération procédurale du monde afin de rendre le jeu imprévisible. Les positions initiales des ennemis sont aussi aléatoires afin de permettre une nouvelle expérience à chaque partie.

En début de partie, le joueur remplit des champs afin de rendre sa partie unique.

Quand il perd, il est directement ramené au menu.