

Table des matières

1 Présentation du sujet	3
1.1 Introduction	3
1.2 Motivations pour le choix du sujet	3
1.3 Notre jeu	3
2 Découverte du moteur de jeu Unity et état de l'art	5
2.1 Un nouvel outil	5
2.2 État de l'art	7
2.2.1 Article numéro 1	7
2.2.2 Article numéro 2	7
2.2.3 Article numéro 3	8
2.2.4 Parallèle entre nos articles et nos choix de développement	8
3 Conception et implémentation des bases de notre jeux	10
3.1 Les scènes	10
3.2 La map	11
3.3 PlayerManager	12
3.4 InputHandler	12
3.5 Ressources et bâtiment	12
3.6 Les unités	13
3.6.1 Les différents types	15
3.6.2 Les différents comportements	15
3.7 La caméra	16
3.8 L'UI et les pop-ups	17
3.8.1 L'UI du jeu	17
3.8.2 L'UI du tutoriel	17
3.8.3 Les pop-ups	17
4 Conception et implémentation de notre intelligence artificielle ennemie	18
4.1 Notre "Behavior Tree"	18
4.2 Conception du comportement des vagues d'ennemis	20
4.2.1 Première vague	20
4.2.2 Deuxième vague	20
4.2.3 Troisième vague	20
4.3 Implémentation du comportement basique de notre IA	20
4.3.1 Agent réactif	20
4.3.2 Comportement de fuite	21
4.4 Implémentation des différents comportements de notre IA	22
4.4.1 La patrouille	22
4.4.2 Le "cri"	23
4.4.3 Les gardiens	24
4.5 Activation des différents comportements via des booléens	24
4.6 Déclenchement des vagues ennemis d'IA	24
5 Gestion de projet	25
5.1 Organisation générale	25
5.2 Participation à la soirée "Into the Game" du 19 Avril 2022	25

5.3	Utilisation de GitHub	25
5.4	Organisation des tâches dans le temps	25
5.5	Améliorations que nous aurions pu apporter à notre gestion	27
6	Résultats et conclusion	28
6.1	Retour sur notre diagramme de Gantt	28
6.2	Rétrospective de notre projet	28
6.3	Suite du projet	28
6.4	Conclusion de notre projet	29
6.5	Remerciements	29
7	Bibliographie	30
8	Annexe	31

1 Présentation du sujet

1.1 Introduction

Dans le cadre du TER, module HAI823I, du second semestre de première année de master informatique, nous avons choisi de travailler sur le développement d'un jeu vidéo. Nous avions le choix concernant le type de jeu et nous avons décidé de faire un jeu de stratégie en temps réel. Notre groupe est composé de Léa Garcia, Arthur Engel (tous deux en Master GL) et de Marie-Lou Desbos (en Master IASD).

1.2 Motivations pour le choix du sujet

Aucun de nous n'est dans le parcours Imagine, qui forme à la programmation de jeux vidéos, mais pour autant nous étions tous les trois très intéressés par la découverte d'un autre domaine de l'informatique. Nous avons décidé de choisir ce sujet d'une part pour nous familiariser avec le monde du développement de jeux vidéos mais aussi pour acquérir de nouvelles compétences que nos master respectifs ne nous apporteront probablement pas.

1.3 Notre jeu

Le type de jeu RTS

Un jeu RTS (Real-Time Strategy, Stratégie en temps réel), est, comme son nom l'indique, un jeu-vidéo de stratégie qui s'oppose au jeu de stratégie tour-par-tour (ou chaque joueur effectue des actions à tour de rôle, comme aux échecs, par exemple). Par RTS, on signifie généralement un jeu où au moins deux joueurs s'affrontent dans un environnement fermé (appelé carte, niveau, ou encore map). Le but est de vaincre les autres joueurs (en fonction des jeux, cela peut consister à vaincre leur armée, détruire un bâtiment d'intérêt, tuer leur roi...).

Pour cela, les joueurs possèdent des unités, à minima des travailleurs et des guerriers. Les travailleurs peuvent récolter des ressources (bois, fer...), et peuvent éventuellement se battre, les guerriers servent uniquement pour le combat. Les ressources servent à construire des bâtiments, et recruter de nouvelles unités pour agrandir son armée. Les bâtiments peuvent servir entre autres, à stocker plus de ressources (scierie, forge..), avoir un rôle de défense (tour de défense, mirador..), recruter de meilleures unités (caserne)...

Cette base peut bien sûr être améliorée, certains jeux rajoutent la notion de technologies à rechercher, qui permettent de créer de meilleurs bâtiments ou unités. On peut créer différents types d'environnement dans la carte, qui modifient le comportement des unités (de l'eau qui ralentirait les unités la traversant, des forêts réduisant la vision, des montagnes infranchissables...). On peut créer des unités plus complexes, qui peuvent attaquer de loin ou en zone, soigner d'autres unités, ou ayant des caractéristiques aléatoires pour diversifier le jeu. On peut rajouter des formations de guerre pour les unités de combat (en pointe, en carré...), et bien d'autres choses, les possibilités sont infinies. C'est un type de jeu très riche, chaque représentant du genre a ses spécificités qui lui permettent de se démarquer.



FIGURE 1.1 – Capture d'écran de Starcraft 2, une des figures de proues des jeux-vidéos de type RTS

Notre jeu, qui s'inscrit dans ce genre, a comme but de survivre à plusieurs vagues d'ennemis, qui ont un comportement différent à chaque vague. Il faut donc réussir à s'adapter à leur comportement en adoptant une bonne stratégie.

2 Découverte du moteur de jeu Unity et état de l'art

2.1 Un nouvel outil

A la recommandation de notre encadrant, nous avons décidé de choisir le moteur de jeu Unity pour faire notre projet. Du fait que nous ne connaissons pas cet outil, la période d'apprentissage initiale a créé un délai dans le développement dont nous n'avions pas totalement anticipé l'ampleur.



FIGURE 2.1 – Le logo d’Unity

Unity est un moteur de jeu-vidéo, édité par Unity Technologies. C'est un outil aidant au développement de jeux-vidéos, implémentant, entre autres, un système de rendu, de physique, de son... Et proposant un éditeur, se voulant intuitif, même pour des non-développeurs.



FIGURE 2.2 – L’éditeur d’Unity, avec notre projet ouvert

L’éditeur propose de nombreuses fenêtres contenant des fonctionnalités utiles, qui sont déplaçables et “dockables”. Parmi les plus importantes on retrouve l’explorateur, qui permet de naviguer dans les fichiers du projet, la hiérarchie, qui permet de voir et modifier la hiérarchie des GameObject (dont nous parlerons un peu plus loin), l’inspecteur, qui permet de voir et paramétriser les Composants de ces GameObject, et la

vue scène, qui permet de visualiser la Scène actuellement ouverte (nous détaillerons également le concept de Scène un peu plus bas).

Pour ce qui est du développement en lui-même, il est centré autour d'un patron architectural appelé Entité-Composant-Système (ECS). C'est un patron que l'on retrouve généralement dans le domaine du développement de jeux-vidéos, qui favorise la composition par rapport à l'héritage, ce qui permet, en partie, de faire fi des difficultés pouvant découler de la programmation orientée objet classique (couplage fort, hiérarchies complexes et contraignantes...).

Une Entité est objet générique, tout élément du jeu (caméra, lumière, personnage, décor, élément d'interface..) est une Entité, à laquelle est rattachée des Composants, qui représentent un aspect de cette Entité (par exemple, pour représenter un joueur, on peut imaginer une entité ayant un ou des composants représentant son apparence, un composant contenant sa zone de collision, un composant détaillant ses caractéristiques, un composant contenant son comportement...). Un des avantages des Composants est leur réutilisabilité et leur modularité, par exemple le Composant contenant une zone de collision décrit dans l'exemple précédent pourrait être utilisé pour n'importe quel élément du jeu, indépendamment des autres entités l'utilisant et des autres composants présents sur l'entité (même si parfois, on souhaite avoir des Composants dépendant d'autres Composants). Un Système, quand à lui, est un processus qui agit sur toutes les Entités possédant le(s) Composant(s) désiré(s), par exemple un système de physique.

Dans le cas d'Unity plus particulièrement, les entités sont nommées des GameObject (et c'est cette appellation que nous utiliserons désormais), qui peuvent être contenus les uns dans les autres, créant une relation parent-enfant. Tout GameObject possède au moins un Composant, nommé Transform, qui représente la position, la rotation, et l'échelle (taille) de ce GameObject relativement à son GameObject parent. Unity propose en outre une grande variété de Composants pour faciliter le développement de jeux, comme des boîtes de collisions, des rendus de maillages et de textures, des caméras... Nous ne détaillerons pas de Systèmes ici car Unity implémente de façon opaque tout ce dont nous avions besoin.

Les scripts programmés par les développeurs (permettant par exemple de gérer les entrées souris et clavier, les intelligences artificielles, la gestion de l'interface..), et peuvent être attachés à un GameObject en tant que Composant, c'est comme cela que l'on rajoute de la logique métier à nos GameObject. Les scripts créés sur Unity héritent par défaut de la classe MonoBehavior, une classe qui implémente notamment deux méthodes, Start et Update, méthodes cœur du scripting sous Unity. Start permet d'appeler des instructions à l'instanciation du GameObject auquel le script est associé (par exemple des initialisations de variables), Update est une fonction qui est appelée à chaque unité de temps (frame ou tick) du jeu (c'est le principe d'une Game Loop). Il existe d'autres méthodes exposées par MonoBehavior, dont nous parlerons plus tard si nous les utilisons, mais Start et Update sont les principales. On n'utilise pas de constructeur dans les scripts Unity, car le cycle de vie du script dépend du GameObject auquel il est rattaché. Il est aussi possible de créer des classes n'héritant pas de MonoBehavior, si on en a besoin (Par exemple pour des classes contenant des variables globales, des classes statiques ou utilitaires, les ScriptableObjects que nous mentionnerons brièvement...)

On peut sauvegarder un GameObject avec ses Composants et ses GameObject enfants afin de le réutiliser, on appelle cela un Prefab. Cela permet de créer des éléments de jeux réutilisables et paramétrables (par exemple un arbre, une unité, un bâtiment etc..). Il est également possible d'instancier programmatiquement des Prefabs dans la Scène en cours du jeu (par exemple un bâtiment créant des unités va instancier le Prefab correspondant à l'unité que l'on souhaite créer). On peut également créer des Prefab Variants, qui sont des Prefab qui dérivent d'une Prefab existante, assimilables à l'héritage en programmation objet.

Unity utilise ce qu'on appelle des Scènes pour contenir un ensemble de GameObject. L'usage qu'il est fait des Scènes vis-à-vis du découpage d'un jeu varie en fonction des développeurs et des projets, certains vont mettre tout un jeu dans la même Scène, d'autres vont mettre chaque élément ("écran") de leur jeu dans une Scène séparée (par exemple, une scène pour le menu, une scène pour chaque niveau, une scène pour les cinématiques, etc...). Les Scènes sont définies de manière assez générique et leur utilisation dépend des besoins et préférences de chacun. On peut l'imaginer comme un monde ou un univers au sein du jeu. Une Scène contient généralement au moins une Caméra principale, pour afficher quelque chose au joueur, et une source de Lumière principale, pour que les éléments de la Scène soient visibles (avec éventuellement une gestion des ombres). Une Scène peut être vue comme le parent des GameObject étant au sommet de

la hiérarchie de ladite Scène, le Transform de ces GameObject représente donc leurs positions, rotations et échelles absolues.

Le moteur Unity utilise le langage de programmation objet C# pour l'écriture de ses scripts. Ce langage était nouveau pour nous au début de ce projet mais comme nous connaissions déjà plusieurs langages de programmation orientés objets (tels que Java, C++...) nous avons facilement pu nous y adapter. La documentation de Microsoft nous a permis de nous familiariser avec la syntaxe du langage et la documentation d'Unity nous a permis d'apprendre à utiliser les concepts propres à Unity comme les fonctions Start, Update, GetComponent, Find, etc.

Parmi les alternatives à Unity, on peut notamment citer Unreal Engine, et Godot, un logiciel libre et open-source.

2.2 État de l'art

Dans le cadre de ce TER, nous avons chacun choisi un article scientifique. Ces articles nous ont permis de faire une partie de l'état de l'art du domaine des jeux vidéos mais nous ont également donné des pistes de réflexion pour le développement de notre jeu et de notre IA.

2.2.1 Article numéro 1

A Bayesian model for plan recognition in RTS games applied to StarCraft

Cet article détaille la conception d'un modèle Bayesien non supervisé de machine learning, permettant à une IA de deviser un plan dans le jeu de stratégie Starcraft, dans le cadre d'un duel entre deux joueurs. Les auteurs déplorent les IA "classiques" généralement employées dans ce type de jeu, car elles ne possèdent pas la capacité de planification ou d'adaptation que pourrait avoir un humain, d'où leur choix du machine learning.

Dans cet article, le modèle se concentre seulement sur une partie de ce qui constituerait une IA complète, spécifiquement : Le choix des bâtiments à construire (bâtiments de production d'unités, bâtiments de défense..) ou des technologies à rechercher (dans Starcraft, il est possible de faire des recherches pour développer des technologies plus poussées, qui permettent par exemple de créer des unités plus puissantes), appelés dans l'article BuildTree et TechTree respectivement.

Pour entraîner le modèle, la fonctionnalité de replay présente dans le jeu a été utilisée. Les replays enregistrent toutes les actions de chacun des joueurs au cours d'une partie. Les chercheurs ont donc récupéré des replays de joueurs de haut niveau, qui ont été analysés afin de créer un modèle pouvant prédire des BuildTree et TechTree optimaux, qui s'adaptent aux actions de l'adversaire.

Une des différences clé de ce type d'IA, est l'approche probabiliste (par opposition à la logique classique) vis-à-vis de ses décisions, qui permet un comportement plus flexible et adaptatif, et une prise de décision efficace même en cas d'incertitude. Une autre différence est l'évolution, une IA développée avec des techniques de machine learning débutera à un niveau très bas, mais s'améliorera au fur et à mesure des parties analysées ou jouées, là où une IA "classique" ne change pas.

L'article conclut en disant que leur modèle produit des prédictions de haute qualité, résistantes au bruit. Est évoqué une éventuelle difficulté restante à surmonter, le cas d'un joueur qui feindrait une stratégie, pour au final en choisir une autre.

2.2.2 Article numéro 2

A review of real-time Strategy game AI

Cet article passe en revue les techniques d'Intelligence Artificielle utilisées pour les jeux vidéo de stratégie en temps réel, en s'intéressant plus spécifiquement à celles du jeu Starcraft. Il est constaté que les

principaux domaines de la recherche universitaire actuelle portent sur la prise de décision tactique et stratégique, la reconnaissance et l'apprentissage de plans.

Dans la prise de décision tactique, on retrouve par exemple l'apprentissage par renforcement, mais aussi l'arbre de jeu, ainsi que d'autres techniques.

Concernant la prise de décision stratégique, il existe la planification au cas par cas, la planification hiérarchique, l'arbre de comportement, la planification d'espace d'état, l'algorithme évolutionniste, l'architecture cognitive et le raisonnement spatial.

La reconnaissance et l'apprentissage de plan se basent sur différents types de plans. Tout d'abord la reconnaissance déductive, la reconnaissance au cas par cas, l'apprentissage par observation et l'apprentissage à partir de démonstrations.

Cet article souligne également la contribution de la recherche académique dans chacun de ces domaines. L'article compare ensuite l'utilisation de l'IA de jeu dans le milieu universitaire et de l'industrie, constatant que la recherche universitaire est fortement axée sur la création d'agents gagnants, tandis que l'industrie vise à maximiser le plaisir de jouer des joueurs. Il est constaté que l'industrie du jeu vidéo n'utilise pas beaucoup les recherches universitaires car c'est soit inapplicable dans la réalité, soit trop long et risqué de l'implémenter dans un nouveau jeu, ce qui met en lumière un domaine pour de possibles recherches : Faire le pont entre le monde industriel et universitaire. Enfin, les domaines du raisonnement spatial, de l'Intelligence artificielle multi-échelle et de la coopération nécessitent d'être plus approfondies, et des méthodes d'évaluation standardisées sont proposées pour produire des résultats comparables entre les études.

2.2.3 Article numéro 3

Characterising and measuring user experiences in digital games

Dans cet article, les auteurs mettent en évidence l'importance de l'expérience des joueurs dans les choix de développement des jeux vidéos mais également la difficulté à mesurer la qualité de cette même expérience. Ils ont cependant trouvé que parler de "flux" et d'"immersion", permettait de bien caractériser ce que les joueurs ressentent lors d'une expérience de jeu.

Le flux est un état psychologique lié au plaisir, qui est maximal lorsque le joueur est complètement absorbé dans son activité. Pour atteindre ce flux maximal il faut rassembler des caractéristiques telles qu'un jeu avec des règles, un besoin de compétences, des objectifs clairs, un besoin de concentration sur une tâche précise, une implication totale, une perte de la notion de temps...

L'immersion est quant à elle, le degré d'implication d'un joueur dans son jeu. Selon ces scientifiques, il existe trois niveaux, l'immersion sensorielle, l'immersion par challenge et l'immersion imaginaire. Elles impliquent respectivement des sensations par rapport au visuel et à la perception du jeu, des défis proposés par le jeu et enfin, des possibilités d'imagination facilitées par la richesse de la narration dans le jeu.

Cet article met en évidence l'importance de l'expérience joueur, et donc de l'augmentation du flux et la présence d'un ou de plusieurs types d'immersion. Les développeurs doivent prendre en compte ces aspects s'ils veulent pouvoir créer une expérience joueur la plus agréable possible.

2.2.4 Parallèle entre nos articles et nos choix de développement

Les trois articles de recherches que nous avons choisi de lire et d'utiliser nous ont guidé dans notre développement.

Initialement, avant de commencer le projet, nous voulions créer une IA très poussée, d'où le temps alloué au développement de celle-ci dans notre diagramme. Nous envisagions d'éventuellement intégrer des techniques de machine learning dans la conception de notre IA. Mais nous avions largement sous-estimé le temps que nous aurions à passer sur le reste du projet. Nous sommes donc restés sur une IA un

peu plus simple qui change de comportement en fonction de conditions, d'états, et de son environnement. Nous avons décidé de nous concentrer sur une IA plus basique afin de correspondre aux attentes du monde industriel du jeu vidéo qui a pour but de créer des IA exigeantes et amusantes mais qui peuvent être vaincues.

Afin de réaliser un jeu correspondant aux attentes classiques de futurs joueurs ou testeurs, nous avons essayé de faire en sorte que le flux de plaisir soit graduel, que le joueur puisse être encadré avec nos règles, ait comme but la survie, puisse montrer ses compétences de réflexion face à l'IA de niveau 3, se concentre dès le début afin de ne pas se laisser submerger d'ennemis. Nous avons fait en sorte que la difficulté soit graduelle et permette de ne pas frustrer ni ennuyer le joueur. Enfin, pour créer une certaine immersion, nous avons choisi l'immersion par challenge qui permet au joueur d'avoir en tête un objectif : survivre aux trois vagues pour gagner.

3 Conception et implémentation des bases de notre jeux

Notre jeu comporte 3 unités différentes. La première est une unité nommée "Worker" qui a pour unique but d'aller chercher des ressources. On la reconnaît dans notre jeu grâce à sa couleur bleue. La seconde est une unité nommée "Warrior" qui est chargée de combattre l'ennemi, et ne peut récupérer de ressources. Ce sont les unités de couleurs violettes. L'ennemi possède son unité, qui sont aussi des "Warrior", mais ceux-ci sont de couleur rouge, et possèdent quelques Composants en plus, nous la détaillerons dans la prochaine partie dédiée à l'IA.

3.1 Les scènes

Notre jeu se divise en trois scènes différentes. Afin de passer d'une scène à l'autre, il faut les rajouter à la liste "scenes in build". Cela va leur attribuer un identifiant qui nous servira lorsque l'on voudra indiquer quelle scène nous souhaitons charger.

Le menu : Dans cette scène, comme son nom l'indique, se trouve le menu. On y a ajouté une image de fond, un titre créé avec l'UI Toolkit dont nous vous parlerons plus en détail à la fin de cette partie, ainsi que trois boutons qui sont représentés par des GameObjects. Chaque bouton est lié à une fonction du script "MainMenu". Le bouton *Jouer* qui lance une partie est lié à la fonction *PlayGame()* qui utilise la fonction *LoadScene()* de la classe *SceneManager*, fournie par Unity, et prend en paramètre l'identifiant de la scène principale *game*, le bouton *Tutoriel* qui mène au tutoriel du jeu, est lié à la fonction *PlayTuto()*, qui utilise aussi la fonction *LoadScene()* avec en paramètre l'identifiant de la scène *Tutoriel* et enfin le bouton *Quitter* qui sert à quitter le jeu en utilisant la fonction *Quit()* de la classe *Application* (fournie par Unity).

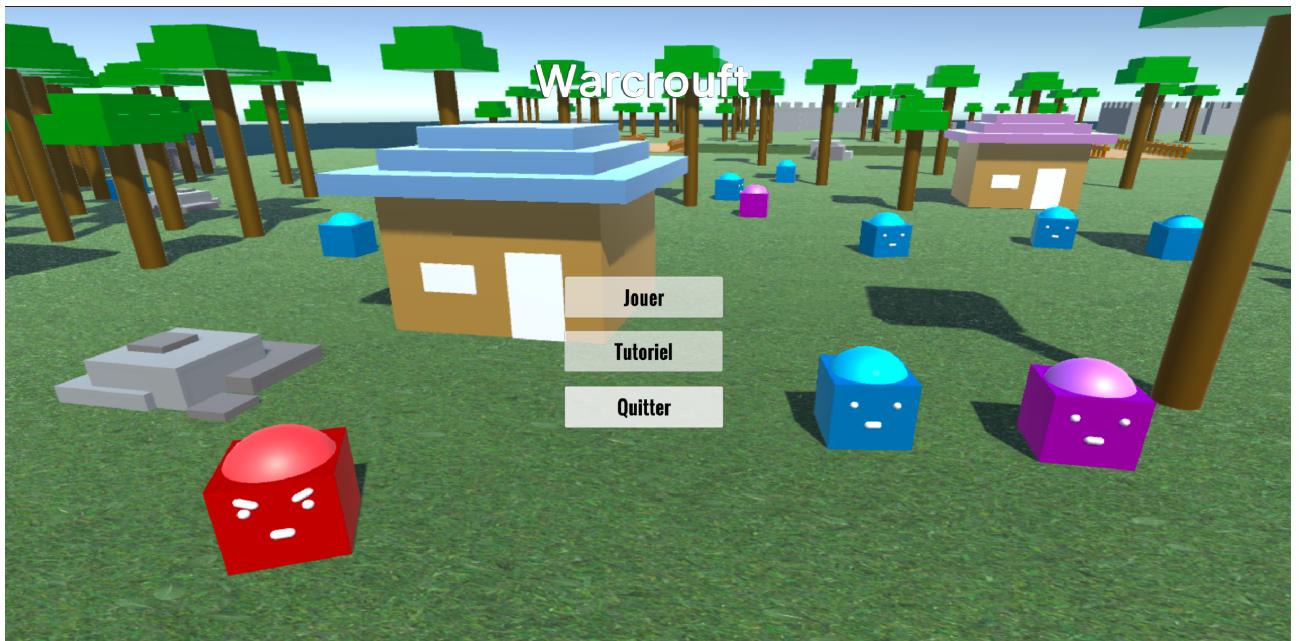


FIGURE 3.1 – Menu de notre jeu

La scène principale *game* : C'est dans cette scène que se passe notre jeu. Elle contient une map, une caméra, une lumière et des unités. Nous allons tout au long de cette partie expliquer plus concrètement le contenu de cette scène.

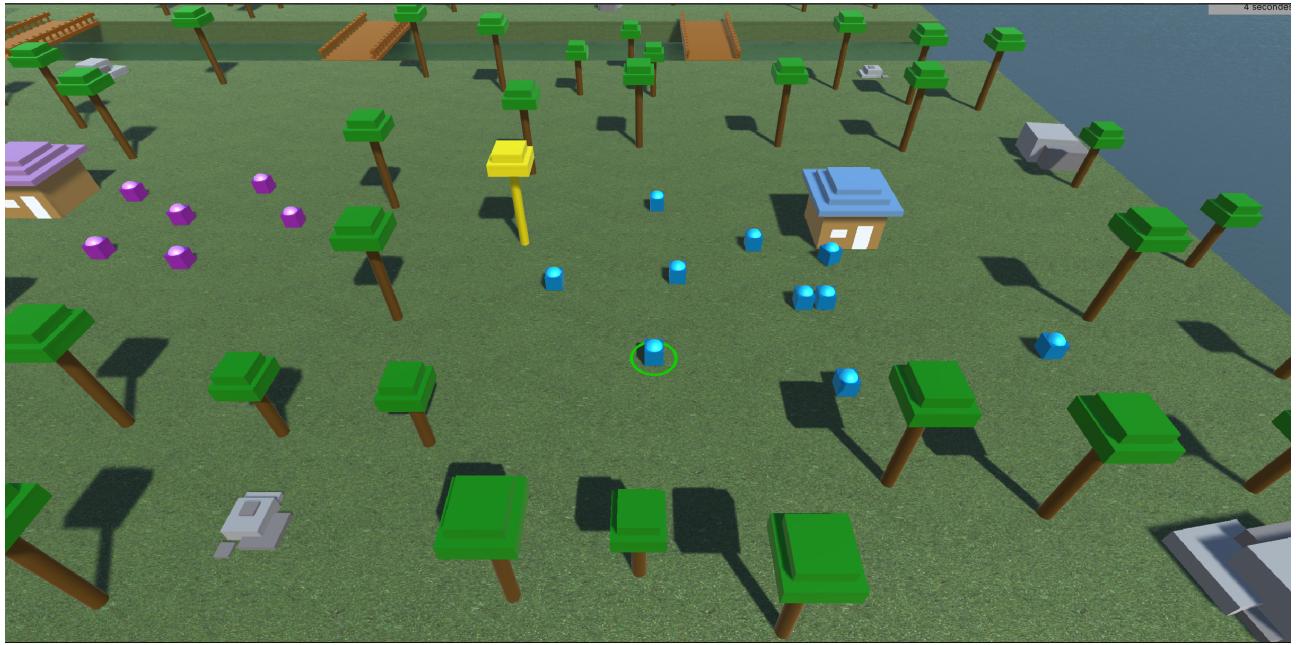


FIGURE 3.2 – Notre jeu, contenu dans la scène "game"

Le tutoriel : Cette scène est identique à la scène principale *game*, à l'exception qu'elle contient des explications sur la manière de jouer. Elle explique comment sélectionner des unités et leur donner des ordres, afin que le joueur puisse prendre en main le jeu avant de commencer à jouer. Aussi, elle ne contient pas d'ennemis, pour que le joueur puisse se concentrer sur la gestion de ses joueurs sans être dérangé. Cette scène se termine automatiquement au bout d'une minute et lance une vraie partie en redirigeant le joueur vers la scène principale.

3.2 La map

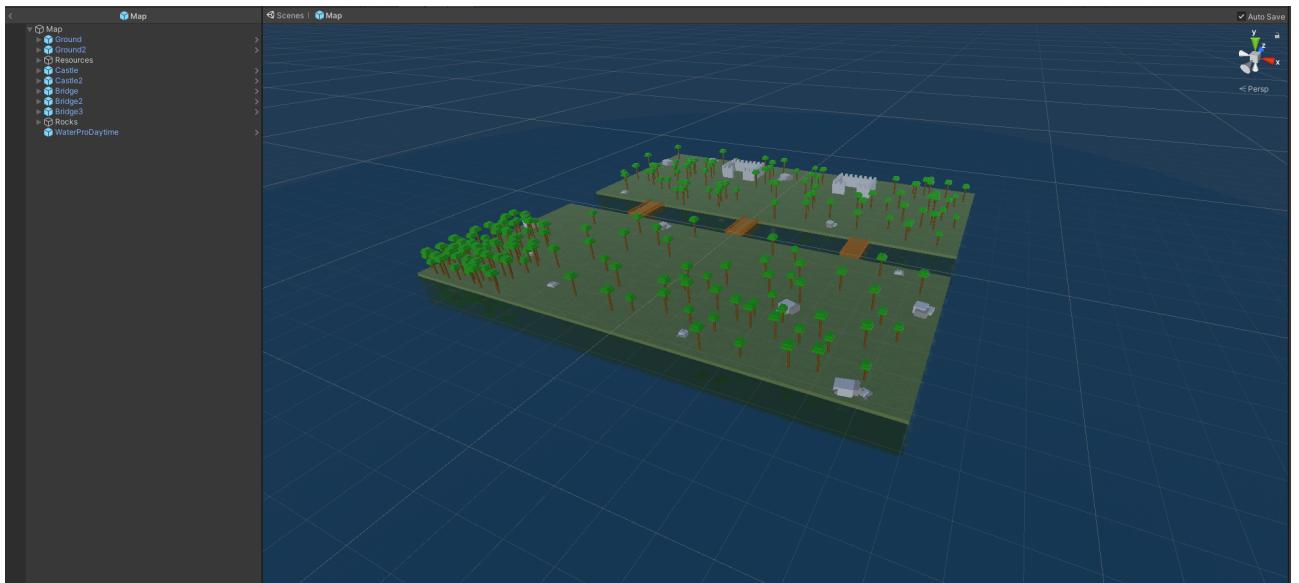


FIGURE 3.3 – L'éditeur de prefab, ouvert sur le prefab de notre map

Le style de notre jeu est plutôt simple et mignon. Les formes sont cubiques ou circulaires. Pour faire la map, nous avons créé un GameObject qui contient tous les éléments du décors de notre jeu (qui sont eux même des GameObject contenant d'autres GameObject pour chaque éléments visuels les composant).

Le sol et les ponts : Le sol est séparé en deux parties distinctes, reliées par trois ponts. Sur une partie se trouve les chateaux, repères des ennemis, et sur l'autre se trouve les bâtiments du joueur.

Les chateaux : Ce sont également des GameObjects. Ils sont liés à un script qui est en charge de créer les ennemis au bon moment durant la partie.

Les arbres : Les arbres sont essentiels à notre jeu, en effet se sont eux qui contiennent la ressource du jeu : le bois. Ce GameObject contient un script contenant la quantité de ressource restante sur un arbre donné, et servant à changer la couleur de l'arbre quand la souris du joueur lui passe dessus, afin qu'il puisse voir plus facilement s'il clique bien sur l'arbre ou à côté.

Le bâtiment des "Workers" : Ce bâtiment au toit bleu est la réserve de bois. Il est lié à un script qui récupère le nombre de bois apporté par les Workers.

Le bâtiment des "Warriors" : Ce bâtiment au toit violet sert à créer des unités de type Warriors une fois qu'assez de bois à été récolté par les Workers. Pour cela il est lié à un script qui regarde le nombre de bois présent dans le stock du joueur. Si ce total atteint 150, il crée un Warrior qui apparaîtra à côté du bâtiment en question.

3.3 PlayerManager

Nous incluons un type de classe, le Manager, très commun dans le milieu du jeu-vidéo. Notre PlayerManager contient les informations de niveau supérieur de notre jeu, c'est à dire les unités de chaque camp, le stock de bois du joueur, le numéro de la vague d'ennemis. Il s'occupe également de lancer la gestion des entrées clavier/souris par le script InputHandler.

3.4 InputHandler

Le InputHandler permet de recevoir les entrées clavier/souris du joueur, et d'agir en conséquence. Elle stocke aussi les unités sélectionnées par le joueur. Elle permet de traduire les entrées clavier/souris du joueur en actions dans le jeu. Le joueur peut sélectionner des unités avec un clic gauche (un simple clic pour sélectionner une unité, ou en maintenant le bouton enfoncé et en faisant glisser la souris pour en sélectionner plusieurs), si l'on fait un clic gauche sur autre chose qu'une unité, cela permet de les désélectionner. Le clic droit sert à faire faire des actions aux unités, en modifiant leur état. Si on clique sur un arbre les Workers sélectionnés iront récolter sur l'arbre, si on clique sur un ennemi les Warriors sélectionnés iront attaquer l'ennemi, si l'on clique ailleurs les unités sélectionnées se dirigeront vers l'endroit cliqué. L'InputHandler récupère les GameObject (arbre, ennemi) ou positions cliquées et les fait passer aux unités sélectionnées pour qu'elles puissent agir en conséquence.

3.5 Ressources et bâtiment

Le joueur dispose d'un bâtiment récoltant le bois, nous avons donc associé un script "MainBuilding.cs" qui s'occupe juste de récupérer le stock de bois, ce stock étant actualisé par chaque Worker qui ramène du bois, dans le script du PlayerManager (A noter que le bois total du joueur est égal au bois contenu dans le bâtiment des Workers, rendant le script de ce bâtiment redondant. C'est parce qu'initialement, nous voulions créer des bâtiments dédiés au stockage des ressources, avec une capacité limitée. Le PlayerManager se serait contenté de compter les ressources conglomérées de tous ces bâtiments). Ensuite, le joueur dispose d'un deuxième bâtiment qui crée des unités de types Warriors. Nous avons associé à ce bâtiment le script "CreationBuilding.cs" qui vérifie en permanence le nombre de bois contenu dans la variable *woodStock* du PlayerManager. Comme on peut le voir dans le code ci-dessous, dès que le stock de bois est supérieur ou égal à 100, un Warrior est créé, ici nous avions mis une boucle while qui permettait d'instancier plusieurs

Warrior à la fois, puis nous nous sommes rendu compte que le jeu devenait trop facile. Nous avons donc baissé le nombre d’itération à 1 en laissant tout de même la boucle pour pouvoir la changer plus tard si besoin

```

21 void Update()
22 {
23     storedResources = PlayerManager.GetInstance().WoodStock;
24     if(storedResources >= 150){
25         PlayerManager.GetInstance().WoodStock = PlayerManager.GetInstance().WoodStock - 150;
26         storedResources = storedResources - 150;
27
28         Debug.Log("Création de 1 warrior ici!");
29         Debug.Log("Utilisation de 200 ressources de bois");
30
31         boucle = 1;
32         Vector3 homeCreation = GameObject.Find("UnitsBuilding").transform.position;
33         while (boucle != 0)
34         {
35             Vector2 randomPos = Random.insideUnitCircle.normalized * 4;
36             Instantiate(myPrefab, homeCreation + new Vector3(randomPos.x, 0, randomPos.y), Quaternion.identity, PlayerManager.GetInstance().playerUnits);
37             boucle--;
38         }
39
40     }
41 }
```

FIGURE 3.4 – Code de la classe CreationBuilding.cs qui s’occupe de créer des unités s’il y a le stock de bois nécessaire

3.6 Les unités

Les unités sont sauvegardées dans des Prefabs. Ces Prefabs ont pour enfant du GameObject au sommet de la hiérarchie des GameObject possédant tous les maillages et textures représentant l’apparence de notre unité (Face, Mouth, LeftEye, RightEye..), ainsi qu’un GameObject possédant boîte de collision de forme sphérique (*ActionCollider*), légèrement plus grande que l’unité, qui lui sert à déterminer si elle est assez proche pour effectuer des actions (récolter, attaquer...). Chaque Prefab possède également un cercle de sélection (*Highlight*), qui est activé et désactivé programmatiquement pour montrer qu’une unité est sélectionnée et actionnable par le joueur, ainsi qu’un TextMeshPro affiché au-dessus de lui afin d’indiquer les points de vie qu’il lui reste (on ne les affiche que lorsqu’il a subi au moins un coup d’ennemi).

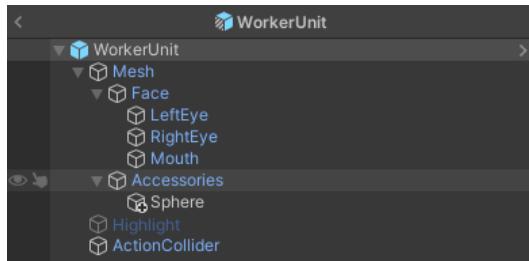


FIGURE 3.5 – La hiérarchie de GameObject du prefab d’une unité, ici la WorkerUnit

Le GameObject au sommet de la hiérarchie (sur l’image ci-dessus, *WorkerUnit*) possède deux composants, un Composant de script d’unité, ainsi qu’un *NavMeshAgent*, un Composant fourni par Unity qui inclut des fonctions de déplacement et de pathfinding. Le *NavMeshAgent* base ses déplacements sur un *NavMesh*, lui indiquant les endroits où il peut se déplacer, et qu’il faut générer sur la géométrie de notre carte, et sur des composants *NavMeshObstacle*, à ajouter sur les éléments de décor que l’on veut que notre agent évite.

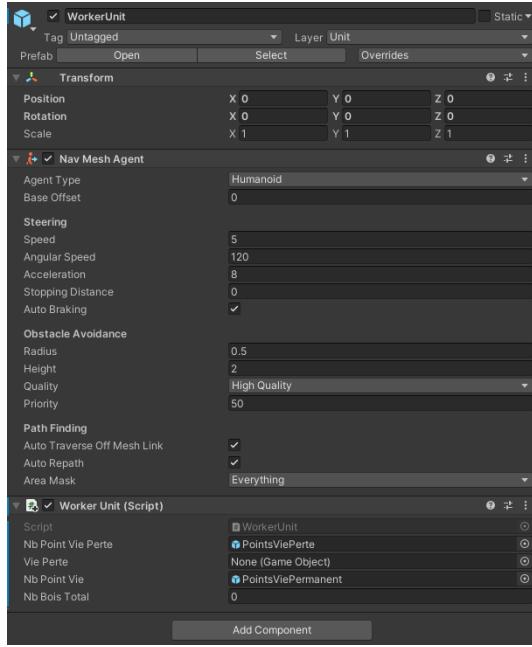


FIGURE 3.6 – Les Composants utilisés pour une unité, ici la WorkerUnit

Les scripts d’unités sont organisés en une hiérarchie très simple. Nous avons les *AbstractUnit*, qui ne sont pas instanciables et contiennent les attributs et la logique métier communs à toutes les unités. Et nous avons ensuite les *WorkerUnit* et les *WarriorUnit* qui héritent de *AbstractUnit*.

Toutes les unités (en somme, la classe *AbstractUnit*) possèdent des attributs contenant leurs caractéristiques, à savoir leur nombre de points de vie actuels (*health*), leur points de vie maximum (*healthMax*), leur coût en ressources, nécessaire pour les invoquer (*cost*), leur puissance d’attaque (*attack*), et leur prénom (*unitName*, non affiché dans le jeu). Les Warriors possèdent également une variable *attackCooldown* et *timeSinceLastAttack*, qui permettent de faire en sorte qu’il y ait un délai entre chaque attaque.

Elles possèdent un booléen indiquant si elles transportent une ressource ou non (*carriesResource*).

Elles possèdent quatre variables permettant de stocker la position de leur base (*home*), la position de l’endroit où elles veulent aller (*target*) (dans le cas d’un déplacement), une cible ennemie à attaquer (*targetEnemy*), et une ressource qu’elles voudraient récolter (*targetResource*).

Elles possèdent une variable stockant leur état (*state*), qui est représenté par une énumération. En effet, nos unités implémentent de façon informelle (non conventionnelle, nous n’utilisons pas le design pattern State) une machine à état.

Et également des variables contenant les *GameObject* et Composants d’intérêt mentionnés précédemment, à savoir l’*ActionCollider*, le *Highlight*, et le *NavMeshAgent*.

Les unités possèdent également une méthode *SubstractHealth*, qui permet de leur enlever de la vie (quand elles se font attaquer), si leur total de vie atteint 0, elles sont détruites par l’appel à une méthode nommée *Die*, qui se charge de nettoyer les références à l’unité avant de la détruire.

Le comportement de l’unité en fonction de son état est écrit dans la fonction *Update* de *AbstractUnit*, la différence entre *WorkerUnit* et *WarriorUnit* se fait ensuite dans l’implémentation des fonctions appelées à partir de *Update*.

Le comportement est tel : une unité peut prendre quatre états différents, *idle* (inactif), *walking* (en marche, allant vers une position), *harvesting* (en cours de récolte), *attacking* (en train d’attaquer).

Une unité est par défaut dans l’état *idle*, dans cet état, elle ne fait rien. Une unité va changer d’état soit par une fonction interne à son script (généralement, pour revenir à l’état *idle* une fois une action terminée), soit par une action extérieure (un humain ou une IA voulant contrôler l’unité).

```

53     protected virtual void Update() {
54         switch(state) {
55             case Globals.unitStates.idle:
56                 break;
57             case Globals.unitStates.walking:
58                 MoveUnitTo(target);
59                 if (targetReached())
60                     SetState(Globals.unitStates.idle);
61                 break;
62             case Globals.unitStates.harvesting:
63                 HarvestRoutine();
64                 break;
65             case Globals.unitStates.attacking:
66                 AttackRoutine();
67                 break;
68         }

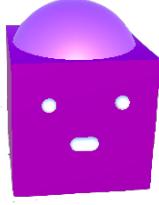
```

FIGURE 3.7 – Fonctionnement de la machine à état de nos unités.

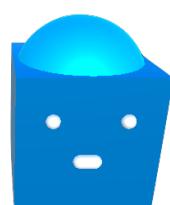
Si l’unité se retrouve dans l’état *walking*, elle va se déplacer vers sa cible, une fois sa cible atteinte (ce que l’on détecte grâce à son *ActionCollider*), elle repasse dans l’état *idle*

Quand l’unité se retrouve dans les états *harvesting* ou *attacking*, les fonctions *HarvestRoutine* et *AttackRoutine* sont appelées, respectivement. C’est ici que les *WarriorUnit* et les *WorkerUnit* diffèrent (ceci et quelques attributs ne changeant pas le comportement du code, comme leur vie maximale) : leur implémentation de ces deux méthodes.

3.6.1 Les différents types



(a) Les Warriors



(b) Les Workers

3.6.2 Les différents comportements

Initialement, nous avions prévu de faire un certain nombre d’unités différentes, et de faire en sorte qu’elles puissent toutes attaquer. Les *WorkerUnit* auraient juste eu une attaque plus faible que les unités guerrières, et nous envisagions que les *WarriorUnit* puissent éventuellement avoir une action particulière (une expression de leur refus de faire autre chose que combattre, par exemple) si on leur demandait de récolter. C’est pour cela que *AttackRoutine* et *HarvestRoutine* sont des méthodes abstraites de *AbstractUnit*, au cas où nous voulions implémenter les deux pour chaque unité. Cependant, comme nous n’avons que deux unités, nous voulions qu’elles soient suffisamment distinctes et uniques, donc présentement, seuls les *WorkerUnit* implémentent une *HarvestRoutine* et seuls les *WarriorUnit* implémentent une *AttackRoutine*

La *HarvestRoutine* des *WorkerUnit* fonctionne comme suit :

- Si l’unité transporte une ressource (*carriesResource* est true)
 - Si l’unité est à portée de la base, elle pose sa ressource, qui est ajoutée au total contenu dans *PlayerManager*
 - Sinon, elle se déplace vers la base.
- Sinon, si elle est à portée de la ressource à récolter
 - Si la ressource n’est pas épuisée, elle récolte, ce qui diminue la quantité de ressource disponible pour cette ressource et change le booléen *carriesResource* de l’unité à true
 - Sinon elle revient à la base
- Sinon, si la ressource n’est pas épuisée, elle se déplace vers la ressource.
- Sinon, elle change sa *target* à *home* et change son *state* en *walking*. (Son état passera donc automatiquement à *idle* quand elle atteindra la base)

La *AttackRoutine* des *WarriorUnit* fonctionne comme suit :

- Si l’ennemi existe (est vivant), on assigne la *target* à la position de l’ennemi, que l’on connaît grâce à *targetEnemy*. On fait cela ici car l’ennemi peut bouger, on a donc besoin d’actualiser cette position. Ensuite...

```

20     public override void HarvestRoutine() {
21         //while(state==Globals.unitStates.harvesting)
22         if(carriesResource) {
23             if(!isHome()){
24                 carriesResource=false;
25                 PlayerManager playerManager = PlayerManager.GetInstance();
26                 playerManager.WoodStock = playerManager.WoodStock + 10;
27             }
28             else
29                 MoveUnitTo(home);
30         }
31
32         else if(targetReached()) {
33             if(targetResource!=null){
34                 Debug.Log("Harvesting!\n");
35                 targetResource.SubtractResource(10);
36                 carriesResource=true;
37             }
38             else
39                 MoveUnitTo(home);
40         }
41
42         else if(targetResource!=null){
43             MoveUnitTo(target);
44         }
45
46         else{
47             SetTarget(home);
48             SetState(Globals.unitStates.walking);
49         }
50     }

```

FIGURE 3.9 – La méthode HarvestRoutine des WorkerUnit

- Si on est à portée de l'ennemi, et que l'on peut attaquer (c'est à dire que *timeSinceLastAttack* est supérieur ou égal à *attackCooldown*), on l'attaque.
- Si on n'est pas à portée de l'ennemi, on se dirige vers celui-ci.
- Sinon, on change le *state* de l'unité vers *idle*.

```

34     public override void AttackRoutine(){
35
36         if(targetEnemy != null) {
37             SetTarget(targetEnemy.transform.position); //On set la target ici et pas dans le input handler car l'ennemi peut bouger (on veut que l'unité s'adapte aux déplacements de l'ennemi)
38             if(targetReached()){
39                 if(timeSinceLastAttack >= attackCooldown){
40                     timeSinceLastAttack = 0f;
41                     targetEnemy.SubtractHealth(attack);
42                 }
43             }
44         }
45
46         else{
47             MoveUnitTo(target);
48         }
49     else{
50         SetState(Globals.unitStates.idle);
51     }
52 }

```

FIGURE 3.10 – La méthode AttackRoutine des WarriorUnit

3.7 La caméra

La caméra est liée à un script nommé "CameraManager", lui-même lié à un GameObject nommé "Main Camera" présent dans la scène *game*. C'est ce script qui va donner un comportement à la caméra durant une partie. Notre caméra se déplace à l'aide de la souris, en la mettant sur un des bords de l'écran, la caméra va se déplacer dans le sens souhaité.

Le code de ce script se trouve uniquement dans la fonction *Update()* car il est important qu'il soit mis à jour à chaque frame. Afin de fixer des limites à la caméra pour qu'elle ne puisse pas aller trop loin dans le monde et qu'elle reste au niveau de la map on a utilisé un Vector2 "cameraLimit" qu'on a initialisé en fonction de notre map. On initialise également une variable de type Vector3 "cameraPosition" avec la position de notre caméra au début du jeu. On a établi la taille de la bordure de l'écran grâce à un float "borderThickness", et la vitesse de la caméra par les float "cameraSpeedUp" pour la vitesse de la caméra quand elle se déplace vers le haut, "cameraSpeedDown" pour le bas, "cameraSpeedLeft" pour la gauche et "cameraSpeedRight" pour la droite. D'abord on regarde sur quel bord de l'écran se trouve la souris du joueur. Pour regarder si elle est sur le bord du haut par exemple, on compare la position de la souris à la hauteur de l'écran moins la taille de la bordure "borderThickness", ce qui nous donne un intervalle en haut de l'écran. Si la position de la souris est dans cet intervalle, alors on met à jour la valeur z de "cameraPosition" en lui donnant la valeur "cameraSpeedUp" multiplié par un timer qui calcule le temps passé depuis le début de la frame afin que la caméra garde cette vitesse tant que la souris se trouve sur le bord de l'écran. On répète l'opération pour chaque bord de l'écran. Une fois qu'on a regardé pour chaque

bord de l'écran si la caméra bougeait ou pas, on applique les changements à "cameraPosition" en utilisant une fonction *Clamp()* présente dans la classe Mathf qui retourne la position de la caméra seulement si elle se trouve dans les limites qu'on lui a données. Et enfin on applique à la caméra du jeu les changements en lui donnant la valeur "cameraPosition".

3.8 L'UI et les pop-ups

Afin de créer notre interface utilisateur nous avons utilisé l'*UI Toolkit*. C'est un outil intégré à l'éditeur d'Unity qui sert à développer des interfaces utilisateurs personnalisées. Pour lier notre UI au jeu, nous avons créé un GameObject "UI" qui contient les scripts nécessaires au bon fonctionnement des différents composants de l'UI.

3.8.1 L'UI du jeu

Lors d'une partie, le joueur peut voir s'afficher en haut à gauche de l'écran le stock de bois qu'il possède. Cette partie de l'UI est gérée dans le script de l'UIController, on récupère le nombre de bois possédé par le joueur dans le PlayerManager et on l'affiche.

L'UI comprend également en haut à droite de l'écran un compte à rebours, dont nous reviendrons sur l'utilité plus tard.

3.8.2 L'UI du tutoriel

Dans le tutoriel, l'UI comprend aussi l'affichage des ressources, géré avec le même script mais le compte à rebours cette fois-ci sert à montrer au joueur combien de temps il lui reste avant la fin du tutoriel. Dans cette UI il y a aussi une partie à droite de l'écran qui contient des images et du texte servant aux explications des commandes au joueur.

3.8.3 Les pop-ups

Afin de faire comprendre au joueur qu'il a perdu ou gagné, nous avons créé deux pop-ups qui apparaissent soit lorsqu'il a éliminé tous les ennemis, soit lorsque toutes ses unités ont été éliminées par l'ennemi. Ces pop-ups sont toutes les deux liées à un même script "CreatePopUp".

Afin de créer la pop-up indiquant au joueur qu'il a gagné, le script regarde s'il n'y a plus de GameObject d'ennemis présents dans la scène et vérifie qu'on en est bien à la dernière vague (nous vous parlerons des vagues plus en détails dans la prochaine partie). Si ces conditions sont remplies, le script instancie la pop-up en créant un GameObject (le prefab de la pop-up) dans la scène. Et idem pour la pop-up indiquant la défaite : le script vérifie s'il n'y a plus de GameObject de type Warrior et Worker alliés dans la scène, et dans ce cas-là on instancie le prefab de la pop-up dans la scène, ce qui la fait apparaître aux yeux du joueur.

Enfin, chaque pop-up est liée à un script : "PopUpController" pour la pop-up de la victoire, et "PopUpLoseController" pour la pop-up de la défaite qui donne un comportement au bouton *Retour au menu* grâce à la fonction *LoadScene()*.

4 Conception et implémentation de notre intelligence artificielle ennemie

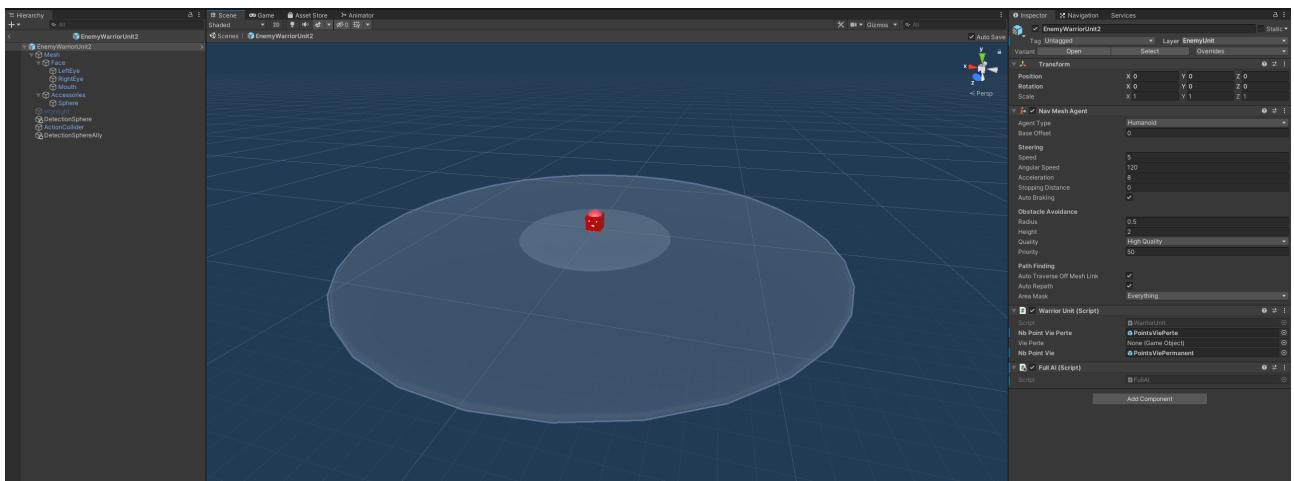


FIGURE 4.1 – Le Prefab des Warrior ennemis. S’ajoute par rapport au prefab des alliés un Composant contenant le script de l’IA, et des sphères de détection

Dans notre jeu, le joueur gagne ou perd. S'il ne dispose plus daucun Warrior ou Worker, alors il a perdu et doit retourner au menu du jeu. S'il réussit à tuer tous les ennemis alors il gagne le jeu. Notre jeu est constitué de trois vagues d'ennemis, ayant un comportement d'IA un peu plus élaboré à chaque fois. Chaque vague est précédée par un affichage sur l'écran du joueur qui indique que la vague est en approche, ainsi qu'un compte à rebours, placé dans l'UI, qui décompte le temps avant la création des ennemis dans leur château. Ce compte à rebours débute avec 15 secondes à chaque approche de vague, nous verrons dans ce chapitre quand et comment ces vagues sont déclenchées. Dès que le jeu commence, le joueur est informé que la première vague d'ennemis est en approche et qu'il lui reste 15 secondes pour former son armée de Warriors.

4.1 Notre "Behavior Tree"

Notre intelligence artificielle fait des choix de comportement en fonction de son environnement. Nous avons donc fait un arbre de comportement afin d'illustrer tous les choix possibles, le voici ci-dessous. Chaque comportement est représenté par un carré avec des bords arrondis, chaque événement comportant ou non des conditions est représenté par une flèche qui va du comportement courant au comportement que l'IA a choisi d'adopter.

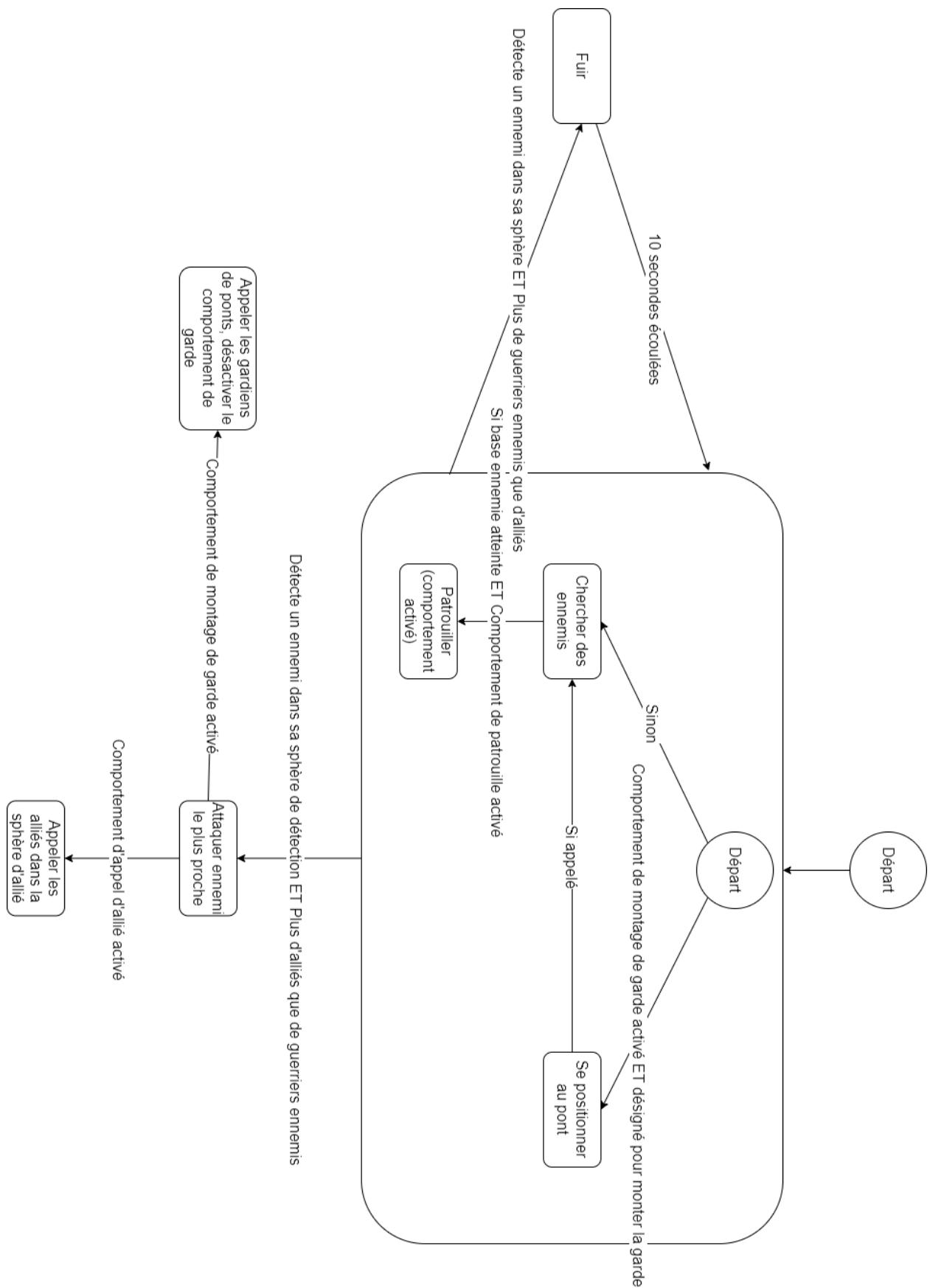


FIGURE 4.2 – Behavior Tree de notre Intelligence Artificielle

4.2 Conception du comportement des vagues d'ennemis

4.2.1 Première vague

Lors de la première vague, l'IA fonctionne de manière assez basique. Il s'agit d'un Warrior, qui est identique à celui des joueurs, sauf que l'on substitue la capacité du Warrior à être contrôlée par la souris à un script d'IA. L'IA contrôle les unités de la même manière que InputHandler contrôle les unités, en donnant des cibles et des états à l'unité qui prend le relais. Les IA possèdent également une sphère de collision qui sert à détecter les ennemis et alliés. Dès qu'une unité dirigée par l'IA est créée, elle enclenche son comportement par défaut : elle cherche des ennemis en allant vers le bâtiment principal du joueur, et attend là-bas une fois celui-ci atteint. A tout moment, si elle détecte un Warrior ou un Worker dans sa petite sphère de détection, elle compte le nombre d'alliés et de Warriors ennemis (Les Workers ne sont pas pris en compte, étant donné qu'ils ne savent pas se battre) : s'il y autant ou plus d'alliés autour d'elle que de Warriors ennemis, elle attaque l'ennemi le plus proche, sinon elle fuit pendant quelques secondes.

4.2.2 Deuxième vague

Lors de la deuxième vague, on rajoute un comportement à l'IA : Quand elle atteint la base ennemie, si elle ne voit pas d'ennemi, elle va patrouiller en carré autour de la base, en élargissant au fur et à mesure son rayon de recherche. Quand elle atteint le rayon de recherche maximal, elle recommence une patrouille, si elle détecte un ennemi, elle agit comme l'IA précédente, et revient à la base adverse une fois l'altercation terminée.

4.2.3 Troisième vague

Lors de la troisième vague, on rajoute deux comportements.
Le premier est un comportement animal, de meute, c'est à dire que chaque unité ennemie, possède une grande sphère de détection autour d'elle, qui va lui permettre d'appeler à l'aide. Lorsque l'IA détecte un ennemi Warrior ou Worker dans sa petite sphère, elle appelle tous les alliés qui sont présents dans sa grande sphère et leur donne sa cible. Ces alliés-là viennent aider s'ils ne sont pas déjà occupés et si c'est le cas, ils finissent leur action avant de venir aider et de se fixer la même cible.
Le deuxième est un comportement de gardien, lors de la création de l'ensemble des unités ennemis, trois d'entre elles sont sélectionnées pour être des gardiens de ponts. Donc leur premier objectif est d'aller se placer au niveau des ponts afin de les garder. Lorsqu'une unité va faire appel à ses alliés présents dans sa sphère, elle va également appeler ses alliés présents au pont, qui quittent alors leur poste pour venir aider.

4.3 Implémentation du comportement basique de notre IA

Dans le jeu, les ennemis sont représentés par des cubes rouges, ils forment une armée de Warriors et n'ont pas de Workers contrairement au joueur. Notre IA peut être interprétée comme étant en majorité un agent réactif, bien qu'elle ne soit pas implémentée de manière scolaire.

4.3.1 Agent réactif

Un agent réactif est une entité (au sens général du terme, pas dans le cadre du patron ECS) autonome, qui évolue dans un environnement. L'agent a une certaine perception de l'environnement, qui va la mener à effectuer une action, action qui changera l'environnement de l'agent, et ainsi de suite.

Dans le cas de notre IA, son environnement consiste en :

- Un endroit où aller chercher des ennemis, que nous désignons comme étant la base des Worker, à savoir le *home* du joueur.
- Les alliés et ennemis présents dans ses sphères de détection (DéTECTÉS par *unitDetectionSphere* et *alliesDetectionSphere*, et contenus dans *enemyUnitsWithinDetectionRange*, *allyUnitsWithinDetectionRange*, et *allyUnitsWithinBigDetectionRange* (ici, BigDetectionRange fait référence aux alliés détectés par la sphère d'appel à l'aide, *alliesDetectionSphere*)).
- Les alliés "gardiens de pont" (contenus dans *backUpList*)

(A noter qu'on ne considère pas la topologie de la carte comme faisant partie de l'environnement de l'IA, car le pathfinding est géré par la couche unité (commune au joueur et l'IA), l'IA elle-même n'a pas

connaissance de celle-ci.)

Toutes les actions de notre IA dépendent uniquement de ces perceptions.

Elle possède également des variables pour retenir des états :

- Le fait d'être en train de fuir ou non (*isRunningAway*)
- Le fait d'avoir atteint la base adverse sans trouver d'ennemis (*enemyBaseReachedAndNoEnemyFound*)
- Un rayon de patrouille, borné, contenus dans les variables *patrolAreaDistance*, *patrolAreaDistanceMin*, *patrolAreaDistanceMax*
- Un numéro, identifiant le coin de son carré de patrouille que l'IA cherche à atteindre (*patrolSquareAreaCornerNumber*)
- Un booléen, qui indique si la direction de patrouille vient de changer (*patrolDirectionChanged*)
- Le fait d'être une unité "gardienne de pont" ou non. (booléen *iambbackup*)
- La position du pont à garder, si l'unité est gardienne de pont (*myBridgePosition*)

Il y a deux choses qui font que notre IA dans son ensemble pourrait être considérée comme n'étant pas purement un agent réactif : la fonctionnalité désignant les agents "gardiens de pont" se situe à un ordre supérieur de celui de l'agent (c'est la classe qui les désigne et non l'agent qui se désigne de façon autonome), et l'appel des gardiens de pont à une portée illimitée, ce qui peut aller à l'encontre d'une philosophie agent "pure", mais qui a du sens dans le cadre d'un jeu de stratégie, où un joueur a une vue d'ensemble sur ses troupes.

4.3.2 Comportement de fuite

Quand notre IA doit fuir, on met *isRunningAway* à true, et elle se dirige vers sa moitié de la carte, nous avons décidé de sa position de fuite de manière arbitraire (idéalement nous aurions implémenter un système de fuite plus intelligent, qu'on récupérerait avec une fonction *getRunAwayDirection* par exemple). Nous avons mis un compte à rebours (Décompté grâce à la méthode *Invoke*, qui permet d'appeler une méthode après un certain temps, ici la méthode *CancelRunAway*, qui met *isRunningAway* à false), qui fait office de "mémoire", qui fait qu'elle continuera à fuir pendant quelques secondes peu importe ce qu'il se passe. Nous avons fait cela pour éviter un comportement où l'IA commencerait à fuir, puis reviendrait immédiatement à la charge une fois les ennemis sortis de sa zone de détection, créant un phénomène d'aller-retour assez peu immersif.

```
233     void runAway() {
234         if (!isRunningAway)
235         {
236             isRunningAway = true;
237
238             Debug.Log("Je cours");
239             unit.SetTarget(new Vector3(0, 0, 55));
240             unit.SetState(Globals.unitStates.walking);
241
242             Invoke("CancelRunAway", 10.0f);
243
244         }
245     }
```

FIGURE 4.3 – Code de la fonction *runAway*

```
280     private void CancelRunAway()
281     {
282         isRunningAway = false;
283         Debug.Log("J'arrete de fuir");
284     }
```

FIGURE 4.4 – Code de la fonction *cancelRunAway*

4.4 Implémentation des différents comportements de notre IA

4.4.1 La patrouille

Le comportement de patrouille est contenu dans la fonction *patrolEnemyBase*, L'IA a un rayon de recherche minimum, un rayon de recherche maximum, et son rayon de recherche effectif (*patrolAreaDistance*, *patrolAreaDistanceMin*, *patrolAreaDistanceMax*). Elle patrouille en carré autour de la position où elle va chercher les ennemis (*home*). Au début son rayon de recherche effectif est identique au rayon minimum.

Pour commencer sa patrouille, l'IA doit trouver la position d'un coin du carré (on commence arbitrairement par le coin supérieur droit), qui est un décalage par rapport à la position où l'IA va chercher ses ennemis, situé à une distance représentée par *patrolAreaDistance*. Pour déterminer la direction de ce décalage (en d'autres termes, choisir le coin du carré que l'on souhaite, en haut à droite, en bas à gauche...), on associe aux axes X et Z un signe positif ou négatif, en utilisant *patrolSquareAreaCornerNumber* pour les déterminer, signes qu'on combine à la *patrolAreaDistance* précédemment évoquée, pour obtenir la position du coin. Additionnellement, on vérifie si la nouvelle position sort de la carte et on la corrige le cas échéant.

Quand elle atteint ce coin, on ajoute 1 à *patrolSquareAreaCornerNumber*, qu'on ramène au modulo 4, on met le *patrolDirectionChanged* à true, ce qui permet de rentrer dans la condition qui calculera le coin suivant, elle se dirigera ainsi vers le prochain coin, et ainsi de suite.

Quand *patrolSquareAreaCornerNumber* revient à 0, c'est à dire quand l'IA finit un tour, elle augmente de 1 son rayon de recherche. Quand son rayon de recherche atteint le maximum, il est remis au minimum.

```
178     void patrolEnemyBase() {
179         if(patrolDirectionChanged){
180             patrolDirectionChanged=false;
181             int xCoordSign;
182             int zCoordSign;
183             switch(patrolSquareAreaCornerNumber) {
184                 case 0:
185                     xCoordSign=1;
186                     zCoordSign=1;
187                     break;
188                 case 1:
189                     xCoordSign=1;
190                     zCoordSign=-1;
191                     break;
192                 case 2:
193                     xCoordSign=-1;
194                     zCoordSign=-1;
195                     break;
196                 default://3
197                     xCoordSign=-1;
198                     zCoordSign=1;
199                     break;
200             }
201             Vector3 targetOffset = new Vector3(patrolAreaDistance*xCoordSign, 0, patrolAreaDistance*zCoordSign);
202
203             Vector3 newTarget = unit.GetHome() + targetOffset;
204             if(newTarget.x > maxXCoord)
205                 newTarget.x = maxXCoord;
206             else if(newTarget.x < minXCoord)
207                 newTarget.x = minXCoord;
208             if(newTarget.z > maxZCoord)
209                 newTarget.z = maxZCoord;
210             else if(newTarget.z < minZCoord)
211                 newTarget.z = minZCoord;
212         }
213     }
```

FIGURE 4.5 – Début du code de la fonction *patrolEnemyBase*

```

213         unit.SetTarget(newTarget);
214         unit.SetState(Globals.unitStates.walking);
215     }
216     else {
217         if(unit.targetReached()) {
218             patrolSquareAreaCornerNumber = (patrolSquareAreaCornerNumber + 1) % 4;
219             patrolDirectionChanged=true;
220         }
221     }
222 }
223
224 //Si on commence un nouveau tour de patrouille
225 if(patrolSquareAreaCornerNumber == 0) {
226     if(patrolAreaDistance < patrolAreaDistanceMax)
227         patrolAreaDistance++;
228     else
229         patrolAreaDistance = patrolAreaDistanceMin;
230 }
231 }
```

FIGURE 4.6 – Suite du code de la fonction patrolEnemyBase

4.4.2 Le "cri"

Comme nous l'avons vu, la troisième vague comporte deux comportements dont le premier qui peut être apparenté à un "cri". Lorsqu'une unité ennemie attaque suite à une prise de décision, elle appelle la fonction *callAllies()* en lui fournissant la cible qu'elle est en train d'attaquer et qui est de type *AbstractUnit*. La fonction *callAllies()* va récupérer à l'aide d'une boucle *foreach*, chacun des alliés ennemis qui sont présents dans sa grande sphère de détection. Pour cela elle parcourt la liste de tous les ennemis, contenue dans le *PlayerManager* et récupère tous ceux dont la position est comprise à l'intérieur de la sphère de détection. Pour récupérer la position on récupère d'abord le transform puis on accède à la position avec "*transform.position*". Lorsqu'on récupère ces unités on les place dans une liste appelée *allyUnitsWithinBigDetectionRange*, que l'on parcourt ensuite avec une deuxième boucle *foreach* afin de faire un appel sur chacune de ces unités. L'appel effectué est *receiveCall()* auquel on fournit toujours la même unité cible. Cette fonction est définie dans *AbstractUnit* et lorsqu'elle est appelée, elle comporte le "state" de l'unité avec la variable "*Globals.unitStates.attacking*", si elles sont égales cela veut dire que l'unité est occupée pour l'instant, sinon l'unité rejoint l'unité appelante pour l'aider en fixant sa propre cible à la cible donnée en paramètre de la fonction. Chacune des unités appelées, répondant à l'appel (non concrètement mais via son comportement), va appeler les unités présentes dans son rayon ce qui entraîne parfois plusieurs appels successifs. Aussi, étant donné que l'on utilise la fonction *Update* pour contrôler les actions de notre IA, lorsque celle-ci attaque et donc appelle ses alliés, elle continue de le faire tant qu'elle attaque, ce qui signifie qu'une unité peut être occupée au début de l'appel, ne l'est ensuite plus et met sa cible comme celle de l'unité appelante ensuite.

```

257 void callAllies(AbstractUnit unitToAttackWithAlly)
258 {
259
260     //créer liste de mes alliés avec la grosse sphère
261     foreach (Transform child in Player.PlayerManager.instance.enemyUnits)
262     {
263         if (alliesDetectionSphere.bounds.Contains(child.position))
264         {
265             allyUnitsWithinBigDetectionRange.Add(child.gameObject.GetComponent<AbstractUnit>());
266             Debug.Log("J'ai ajouté un allié à la liste!");
267         }
268     }
269
270     foreach (AbstractUnit ally in allyUnitsWithinBigDetectionRange)
271     {
272         Debug.Log(unit.gameObject.GetComponent<AbstractUnit>() + " :J'appelle mon allié pour qu'il vienne m'aider :", unitToAttackWithAlly);
273         if(ally != null)
274         {
275             ally.receiveCall(unitToAttackWithAlly);
276         }
277     }
278 }
```

FIGURE 4.7 – Code de la fonction *callAllies*, appelée par l'unité ennemie qui attaque

```

164     public void receiveCall(AbstractUnit unitToAttackWithAlly)
165     {
166         Debug.Log("J'ai reçu un appel à l'aide");
167         if (state == Globals.unitStates.attacking)
168         {
169             if(targetEnemy == unitToAttackWithAlly)
170             {
171                 Debug.Log("J'arrive !");
172             }else{
173                 Debug.Log("Je suis déjà en train d'attaquer un autre ennemi, je ne peux pas (encore) t'aider");
174             }
175         }else
176         {
177             Debug.Log("J'arrive pour t'aider !");
178             SetTargetEnemy(unitToAttackWithAlly);
179             SetState(Globals.unitStates.attacking);
180         }
181     }
182 }
183 }
```

FIGURE 4.8 – Code de la fonction receiveCall, appelée sur l’unité présente dans la grande sphère de l’unité appelante qui est en train d’attaquer

4.4.3 Les gardiens

Le deuxième comportement ajouté lors de la troisième vague est celui des "gardiens" puisqu'il permet de placer une unité ennemie à chaque pont. Lorsqu'on instancie tous les ennemis, on récupère dans la liste (toujours contenue dans le PlayerManager), trois ennemis que l'on place dans une liste appelée *backUpList*. On crée également une liste contenant trois positions, une pour chaque pont appelée *backUpListBridgesPositions*. Lorsqu'un ennemi décide de fuir avec la fonction *runAway()*, il appelle également la fonction *callBackup()* qui permet, exactement comme le cri vu précédemment, d'appeler sur chaque gardien contenu dans la liste *backUpList*, une fonction nommée *receiveCallBackup()*. Cette fonction permet en quelque sorte de "réveiller" les gardiens et de les faire venir vers le camp des joueurs pour attaquer et reprendre un comportement normal car contrairement aux autres comportements, le booléen est désactivé dans l'appel.

4.5 Activation des différents comportements via des booléens

Afin d'utiliser un Prefab unique pour tous nos niveaux d'IA, nous avons fusionné les scripts des différents comportements que nous avons vus, et nous l'avons mis dans un script nommé "FullIA.cs", les comportements sont activables via des booléens statiques contenus dans le script.

Chaque comportement est contrôlé via un booléen, nous avons le booléen "enablePatrolBehavior", le booléen "enableAllyCallingBehavior" et enfin le booléen "enableBackupBehavior". On vérifie ensuite par des conditions dans notre script d'IA, si chaque comportement est activé ou non.

4.6 Déclenchement des vagues ennemis d'IA

Afin de déclencher des vagues d'ennemis, dans le script UIController.cs, nous avons mis en place des conditions qui vérifient dans quel vague le joueur se trouve et le nombre d'ennemis encore présent dans le jeu. S'il a tué tous les ennemis, on affiche devant son écran l'arrivée de la prochaine vague, on lance une co-routine pour faire l'affichage du compte à rebours et on modifie l'affiche. Ensuite lorsque la co-routine est sur le point de se terminer (dernier appel quand time==0), on modifie le numéro de la vague, ce qui déclenche le script de création d'ennemis IA que nous allons voir.

Afin de répondre au déclenchement d'une vague, nous avons utilisé un script de création qui reçoit le prefab d'une unité ennemie et s'occupe d'en instancier, il s'agit du script "CreateEnemyUnits.cs". Dans ce script, trois conditions s'occupent chacune de détecter si on vient d'entrer dans une nouvelle vague en récupérant la vague de la variable "numeroVague" du PlayerManager. Lors d'un changement de vague, une des trois condition est remplie, et une boucle while instancie et ajoute sur la map des unités ennemis puis s'occupe de leur donner le comportement associé en mettant certains booléens à "true".

5 Gestion de projet

5.1 Organisation générale

Dès la première semaine, nous avons convenu d'un rendez-vous hebdomadaire avec notre encadrant le mercredi en fin d'après-midi. Ces réunions nous permettaient de discuter avec notre encadrant de ce que nous avions réalisé pendant la semaine, ainsi que de lui présenter les idées que nous avions pour la suite du développement de notre jeu et échanger sur ces idées.

5.2 Participation à la soirée "Into the Game" du 19 Avril 2022

Comme chaque année, dans le cadre des différents TER de développement de jeux vidéos du master Imagine, une soirée de démonstration a été organisée. Nous avons été invités à cette soirée afin de présenter une première version de notre projet, donner la possibilité aux visiteurs de tester notre jeu et de nous donner leur avis. Afin de pouvoir présenter une première version jouable nous avons dû terminer certaines tâches de notre Gantt en avance comme l'élaboration d'une IA basique mais suffisante pour représenter l'ennemi du joueur, un pop-up fonctionnel pour pouvoir recommencer les parties etc..

Au cours de cette soirée nous avons reçu des avis par rapport au gameplay, au design, à la difficulté de notre jeu. Nous avons également pu échanger avec eux concernant la suite du projet et principalement comment nous voulions développer notre IA. Nous avons ainsi pu inclure les avis les plus pertinents et poursuivre le développement de notre IA en conséquence.

5.3 Utilisation de GitHub

Pour pouvoir travailler tous les trois simultanément nous avons utilisé le gestionnaire de versions GitHub, nous avons mis en place un fichier `.gitignore` afin d'éviter au maximum les conflits, car Unity utilise de nombreux fichiers temporaires générant du bruit. Aussi, certains fichiers (notamment les scènes), mènent à des conflits de merge insolubles. En conséquence, nous avions comme règle de ne pas toucher (ou du moins ne pas enregistrer) la scène sans prévenir les autres afin d'éviter au maximum ces conflits et de perdre tout le travail réalisé. Malgré tout, nous n'avons pas pu empêcher tous ces problèmes, qui nous ont ralenti au début avant que nous nous y habituions.

5.4 Organisation des tâches dans le temps

Nous avions déjà réalisé un diagramme de Gantt lors de l'écriture de notre feuille de route, cependant celui-ci s'arrêtait au 25 Avril, or le rendu final du TER est prévu pour le 23 Mai. Nous avons donc pendant le développement du projet, mis à jour notre Gantt, adapté les tâches restantes en fonction de celles qui avaient déjà été réalisées et nous avons ajouté les tâches qui manquaient dans le premier Gantt. Nous avons également intégré la soirée Into The Game et les modifications nécessaires suite aux retours obtenus lors de cet évènement.

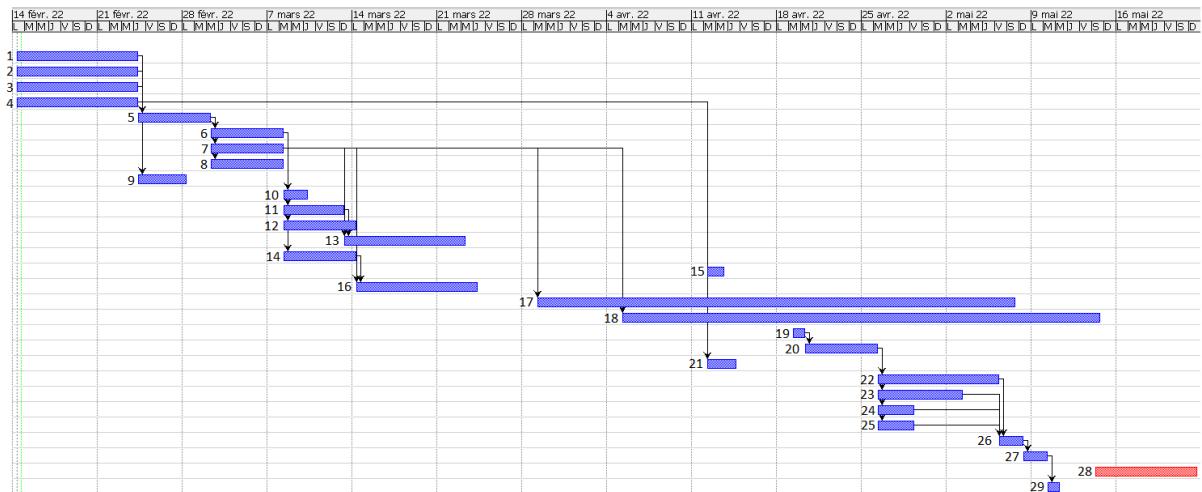


FIGURE 5.1 – Diagramme de Gantt

		Nom	Durée	Début	Fin	Prédécesseur
1		Mise en place de l'environnement	10 jours	14/02/22 09:00	24/02/22 09:00	
2		Map affichable basique	10 jours	14/02/22 09:00	24/02/22 09:00	
3		Personnage déplaçable (graphismes minimaux)	10 jours	14/02/22 09:00	24/02/22 09:00	
4		Menu basique	10 jours	14/02/22 09:00	24/02/22 09:00	
5		Architecture/ Conception du jeu (sauf IA)	6 jours	24/02/22 09:00	02/03/22 09:00	
6		Ajout ressources (logique métier)	6 jours	02/03/22 09:00	08/03/22 09:00	5
7		Ajout classes de personnage (récolteur, soldat, ennemi) (logique métier)	6 jours	02/03/22 09:00	08/03/22 09:00	5
8		Ajout bâtiments (logique métier)	6 jours	02/03/22 09:00	08/03/22 09:00	5
9		Gestion de la caméra	4 jours	24/02/22 09:00	28/02/22 09:00	2
10		Interface utilisateur (affichage ressources)	2 jours	08/03/22 09:00	10/03/22 09:00	6
11		Interface recrutement/construction	5 jours	08/03/22 09:00	13/03/22 09:00	7
12		Assets basiques (personnages, bâtiments, environnement)	6 jours	08/03/22 09:00	14/03/22 09:00	7
13		Implémentation recrutement/construction (choisir où construire/recruter, faire apparaître l'entité, etc..)	10 jours	13/03/22 09:00	23/03/22 09:00	7;11
14		Interaction entre personnages et décor (récolte ressources, construction)	6 jours	03/03/22 09:00	14/03/22 09:00	6;7
15	▣	Création et mise en place des pop-ups	2 jours	12/04/22 08:00	13/04/22 17:00	
16	▣	Interaction entre personnages et personnages (combat)	10 jours	14/03/22 09:00	24/03/22 09:00	7;14
17	▣	Conception de l'IA	40 jours	29/03/22 08:00	07/05/22 17:00	7
18	▣	Programmation de l'IA	40 jours	05/04/22 08:00	14/05/22 17:00	7
19	▣	Into The Game	1 jour	19/04/22 09:00	20/04/22 09:00	
20		Faire les modifications après Into The Game	6 jours	20/04/22 09:00	26/04/22 09:00	19
21	▣	Construction d'un vrai menu	3 jours	12/04/22 08:00	14/04/22 17:00	4
22		Ajout de vagues d'ennemis ayant des comportements d'IA différents	10 jours	26/04/22 09:00	06/05/22 09:00	20
23		Enrichissement de la map	7 jours	26/04/22 09:00	03/05/22 09:00	20
24		Ajout des affichages concernant les vagues (chrono et texte)	3 jours	26/04/22 09:00	29/04/22 09:00	20
25		Ajout d'un tutoriel dans une nouvelle scène	3 jours	26/04/22 09:00	29/04/22 09:00	20
26		Test du jeu (build temporaire)	2 jours	06/05/22 09:00	08/05/22 09:00	25;23;22;24
27		Modification avec phase de tests par de vrais utilisateurs	2 jours	08/05/22 09:00	10/05/22 09:00	26
28	▣	Rapport	9 jours	14/05/22 08:00	22/05/22 17:00	
29		Faire le build du jeu	1 jour	10/05/22 09:00	11/05/22 09:00	27

FIGURE 5.2 – Tâches du diagramme de Gantt

5.5 Améliorations que nous aurions pu apporter à notre gestion

Le projet terminé, et au vu des problèmes rencontrés, nous avons dégagé quelques améliorations à apporter dans nos futurs projets qui nous semblent pertinentes :

- Un plus gros accent sur certains principes de développement, à la SOLID, KISS, DRY.. Et sur les bonnes pratiques de manière générale.
- Dans la continuité du point précédent, une meilleure communication sur le style de code à adopter, et sur les principes architecturaux que nous usitons ? ? ? ? ?, c'est-à-dire tout ce qui est conventions de nommage, séparation des préoccupations, définir plus précisément le rôle et la hiérarchie de nos classes.
- Penser plus à la modularité, et prendre le temps refactoriser notre code dès que possible, pour éviter la dette technique au fil du projet. Il y a beaucoup de solutions "temporaires" ou d'approximations qui nous ont ralenti par la suite, il y a également des bouts de code obsolètes, ou non utilisés car non terminés.
- Éventuellement, des revues de code.
- Une recherche plus extensive du domaine en amont, pour avoir une meilleure compréhension des pratiques communes.

Certains problèmes restaient malgré tout durs à éviter, nous nous sommes laissés emporter dans une course à la fonctionnalité, notamment pour rester dans nos objectifs de Into the Game, rendant la conception de notre jeu perfectible. Cependant, fort de notre nouvelle expérience, nous pensons être en mesure de développer des jeux conçus de manière beaucoup plus robuste à l'avenir.

6 Résultats et conclusion

6.1 Retour sur notre diagramme de Gantt

Par rapport à l'organisation prévue initialement, nous pensons avoir plutôt réussi à suivre les délais imposés et l'ordre des tâches que nous avions défini. Nous avons pu rajouter au cours du projet, des tâches qui n'étaient pas présentes au début car nous n'avions pas encore eu ces idées. Nous avons eu une contrainte de plus avec l'évènement Into The Game et cela nous a poussés à ne pas prendre de retard et arriver avec une version jouable de notre jeu. Certains aspects ne sont pas aussi développés que l'est la vision que nous en avions initialement, mais nous sommes dans l'ensemble satisfaits.

6.2 Rétrospective de notre projet

Alors que nous avancions sur notre projet, nous avons découvert ou pensé à des concepts et manières de faire qui auraient pu améliorer la qualité et l'avancée de celui-ci. Mais en raison de dette technique, et de délais à respecter, nous n'avons pas tout essayé. Nous nous doutons également qu'il y a beaucoup de choses que nous ne connaissons toujours pas, et que certaines de nos implémentations ne seraient pas considérés comme "correctes" par des gens du milieu.

Nous aurions pu modulariser plus le code. Par exemple, on pourrait considérer que la vie d'un joueur et la quantité de ressource d'un arbre sont équivalents sémantiquement, les deux étant un total qui, quand il atteint 0, mène à la destruction de l'entité qui y est associée. On pourrait alors faire un composant séparé "Vie" qu'on attacherait aux entités en ayant besoin, ce qui éviterait le dédoublement du code (principe DRY). De la même manière, on pourrait faire un composant "Sélectionnable", pour toute entité qu'on pourrait sélectionner (une unité, un bâtiment, un arbre...), pour découpler ce comportement de ces entités (séparation des préoccupations). De manière générale, beaucoup de nos classes semblent pouvoir être épurées et découpées. Nous aurions pu utiliser certaines fonctionnalités d'Unity ou du C# que nous ne connaissions pas initialement : les ScriptableObjects, qui permettent de contenir des données indépendantes des instances de classes, les propriétés, une manière élégante d'encapsuler des attributs avec leur getters et setters, éditables dans l'éditeur, ou encore les machines à états visuelles, intégrées à Unity, qui auraient pu être pratiques pour le développement de notre IA... Dans l'ensemble, nous aurions été aidé par une plus grande culture sur ce qui se fait usuellement pour résoudre des problèmes communs (quel composant utiliser pour faire telle chose, comment l'utiliser de façon optimale, etc...). Des design patterns du jeu-vidéo, en quelque sorte.

6.3 Suite du projet

Avec plus de temps, nous aurions pu implémenter plus de vagues d'ennemis, avec différentes IA. Nous avions pensé rajouter à notre IA un premier comportement qui lui permettrait de pouvoir "voir" ses ennemis au loin (les unités du joueur), et de les poursuivre tant qu'ils étaient dans son champs de vision. Nous avons aussi eu l'idée de donner un comportement de type fourmi de Malaisie à notre IA, c'est à dire que si elle se retrouve encerclée et voit qu'elle va forcément mourir, elle explosera et ferait des dégâts aux unités du joueur qui l'entourent. Un dernier comportement, mais bien plus difficile à implémenter, aurait été également un comportement de fourmi, qui permet à une unité ennemie (une IA) en fuite, de laisser derrière elle des traces pour indiquer sa fuite. Ainsi, toutes les unités alliées passant par le même chemin auraient pu recevoir cette information et adapter leur comportement en fonction de celle-ci.

Nous avions également pensé à ajouter du son afin de rendre le jeu plus vivant, comme par exemple du bruit de bois lorsque les workers en ramassent ou encore du bruit lorsque les ennemis et les warriors se battent. Une autre idée était d'agrandir notre map et la diversifier afin de pouvoir augmenter le niveau de difficulté en donnant la possibilité aux IA comme aux joueurs d'adopter des stratégies différentes. Avec toujours plus de temps nous aurions aimé ajouter d'autres types de ressources comme de l'or, de la pierre. Nous aurions aimé ajouter des types de warriors ayant des attaques différentes comme par exemple des attaques de loin, des attaques avec des armes. Nous avons cependant choisi de nous concentrer sur la conception et l'implémentation de notre IA, tout en continuant de maintenir et améliorer les possibilités de jeux déjà présentes.

6.4 Conclusion de notre projet

En conclusion, ce projet nous aura permis à tous les trois, venant de parcours différents, de découvrir un peu le contenu de la formation Imagine et le monde du développement de jeux vidéos. Nous avons pu mettre en pratique des connaissances en programmation, apprises depuis notre première année à la faculté, dans le cadre d'un projet qui ne s'inscrit pas forcément dans la continuité de notre parcours mais qui nous aura beaucoup apporté. Nous avons appris à utiliser Unity, développé nos connaissances avec GitHub, découvert un nouveau langage de programmation, continué d'apprendre à gérer le développement de projets dans le temps et ce sont des savoir-faire qui nous seront forcément utiles pour notre stage et pour notre futur travail.

6.5 Remerciements

Nous tenons à remercier notre encadrant Mr. Simon Besga qui nous a guidés tout au long du projet et à travers notre découverte du monde des jeux vidéos, qui a su nous expliquer les attentes d'un tel projet, et nous a donné de bons conseils pour tenir les délais surtout sur la fin du projet.

7 Bibliographie

Documentation Unity

<https://docs.unity.com/>

Documentation C# de Microsoft

<https://docs.microsoft.com/fr-fr/dotnet/csharp/>

Forum Unity3D France

<http://www.unity3d-france.com/unity/phpBB3/index.php?sid=61bc06ffd24d313a2d60063ed20c3e6e>

Forum Answer Unity

<https://answers.unity.com/questions/index.html>

Forum Answer Unity

<https://answers.unity.com/questions/index.html>

Robertson G. & Watson I., "A Review of Real-Time Strategy Game AI", in *AI Magazine*, Vol. 35, No. 4, Winter 2014, pp. 75-104.

<https://ojs.aaai.org/index.php/aimagazine/article/view/2478>

IJsselsteijn W. A. & De Kort Y. & Poels K. & Jurgelionis A., "Characterising and measuring user experiences in digital games", in *In R. Bernhaupt, M. Tscheligi*, 2007

https://www.researchgate.net/publication/228626547_Characterising_and_Measuring_User_Experiences_in_Digital_Games

Synnaeve G. & Bessiere P., "A Bayesian model for plan recognition in RTS games applied to StarCraft", in *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011

<https://hal.archives-ouvertes.fr/hal-00641323/document>

8 Annexe

Mode d'emploi pour installer Unity et ouvrir le jeu

- 1 Afin de télécharger Unity, il faut installer UnityHub. Pour cela, rendez vous sur <https://unity.com/fr/download>.
- 2 Une fois téléchargé, allez dans vos fichiers et ouvrez votre fichier "UnityHubSetup".
- 3 Suivez les instructions afin d'installer UnityHub, et une fois installé, ouvrez le.
- 4 Unity Hub va vous proposer d'installer Unity Editor 2021.3.3f1, ne l'installez pas, cliquer sur *Skip installation*.
- 5 Une fois dans Unity Hub, cliquer sur le bouton *Install Editor*, puis choisissez la version 2020.3.*f1.
- 6 Une fois l'éditeur installé, si vous n'avez pas encore téléchargé le dossier de notre jeu, téléchargez le, puis allez dans l'onglet "Project", cliquez sur *Open*, et cherchez le dossier du jeu dans vos dossiers.
- 7 Unity Hub va peut être vous dire que la version de l'éditeur utilisé dans notre projet n'est pas installée. Dans ce cas, cliquez sur *Choose another Editor version*, choisissez celle que vous venez d'installer, puis cliquez sur *Open with 2020.3.*f1*, puis sur *Change version*
- 8 Allez dans l'onglet "Projects" et cliquez sur *warcrouft*. Un message d'alerte va s'ouvrir, cliquez simplement sur *Continue*.
- 9 Et voilà ! Vous êtes dans l'éditeur de notre jeu. Si vous voulez ouvrir nos scripts, allez dans l'onglet "Project", puis dans le dossier "Assets" > "_Scripts", et cliquez sur le script que vous voulez voir, il s'ouvrira automatiquement avec votre éditeur de texte par défaut.