KaizerWald
Tower Defense

Table des matières

1 Analyse préliminaire 4
1.1 Introduction 4
1.2 Objectifs 5
1.2.1 Général 5
1.2.2 Camera 5
1.2.3 Ennemis 5
1.2.4 Tourelles 5
1.2.5 Aspect Techniques 5
1.3 Planification initiale 6

2 Analyse / Conception 7
2.1 Concept 7
2.1.1 Maquette de l'interface Utilisateur 7
2.1.2 Conception : Construction d'une tourelle : 7
2.2 Stratégie de test 8
2.3 Risques techniques 9

3 Réalisation 9
3.1 Dossier de réalisation 9
3.2 Semaine 2 : Refactor du Planning 10
3.2.1 Feature Inutile : Problème dans la définition des acteurs 10
3.2.2 Une Conception trop détaillé 11
3.2.3 Conclusion après ces deux semaines de « perdu » : 12
3.3 Tourelles : 13
3.3.1 Système de Tir : Première Solution (fonctionnelle) 13
3.3.2 Amélioration Future : 14
Solution plus élégante mais qui ajoute en complexité mais sera essentielle en terme de mécaniques de gameplay (développé plus loin) 14
3.4 Système de Grilles 15
3.4.1 Généralité 16
3.4.2 Fonctionnalité de la grille partitionnée 17
3.4.3 Interactions entre les grilles : 17
3.4.4 IGridHandler<T, GenericGrid<T>> : 17
3.4.5 IGridSystem : 17
3.4.6 Problème : 2 Grilles – 2 Tailles de cellule 18
3.5 Pathfinding / Recherche de Chemin 22
3.5.1 FlowField : 22
3.5.2 AStar : 25
3.6 Gestion des entités dynamiques (ennemies) 26
3.6.1 Update des entités 26
3.7 Erreurs restantes 28
3.8 Liste des documents fournis 30

4 Conclusions 30
4.1 Objectifs atteints / non-atteints 30
4.2 Points positifs / négatifs 31
4.3 Difficultés particulières 31

4.4 Possible Evolution 32

5 Annexes.....32
5.1 Résumé du rapport du TPI / version succincte de la documentation..... 32
5.2 Sources – Bibliographie..... 32
5.3 Table des Illustrations..... 33

1 Analyse préliminaire

1.1 Introduction

Le projet consiste en la création d'un Tower Defense sur Unity dans le but de mettre en place les éléments nécessaires au futur TPI.

Ce projet a été convenu afin de pouvoir mettre en place les modules bases pour la création d'un RTS qui sera le futur projet TPI. Notamment le management des grilles et du pathfinding des intelligences artificielles.

Ce projet utilise une bibliothèque personnelle que je maintiens depuis que j'ai commencé sur Unity (il y a 1 an et demi) contenant des utilitaires pour la manipulation de grilles et des extensions propres à Unity.

Synthèse du Projet :

Dans ce projet, le jeu comprendra

- Déplacement sur un terrain défini en vue du dessus (vue RTS)
- Construction de tourelles via une interface graphique
- Sélectionner des tourelles pour accéder à leurs améliorations
- Système de vague (les ennemies apparaissent)
- Compteur de vie et de ressources.

1.2 Objectifs

1.2.1 Général

- Le joueur a un nombre de point de vie défini au début de la partie
- Le joueur apparaît tout en bas du terrain, l'objectif à défendre lui est présenté (sous la forme d'un cercle au sol.
- Le point de spawn des ennemis est lui aussi indiqué via un une zone en rouge, cette dernière sera à l'autre extrémité du terrain (en haut du terrain).
- Le joueur a un temps de préparation avant que le round ne commence
- Le joueur a au début une somme de jeton de base pour quelques constructions
- Le joueur perd une vie quand un ennemi arrive à destination
- Si le joueur n'a plus de vie, le jeu se termine
- Gagne un jeton quand un ennemi est tué par une tourelle

1.2.2 Camera

- Définie la vue du joueur
- Rotation horizontal illimité
- Rotation vertical limité à -45° et 30°
- Limitation du mouvement (translation XY) pour éviter que la caméra aille hors du terrain et éviter qu'elle aille en dessous

1.2.3 Ennemis

- Apparaissent au début de chaque Round uniquement.
- Le nombre d'ennemi augmente à chaque round
- Les ennemis ne font qu'avancer et convergent vers une destination précise
- Suivent un chemin défini par un « FlowField » qui les maintient sur un chemin défini
- Quand un ennemi arrive à destination, il disparaît et le joueur perd une vie.

1.2.4 Tourelles

- Construite par le joueur
- Coute des jetons pour être construites
- Peuvent être améliorer contre des jetons

1.2.5 Aspect Techniques

- Architecture du code (SOC, POO, …)
- Au moins un test unitaire est implémenté, à jour et pertinent
- Mise au point du pathfinding avec évolutivité
- Respect de conventions de nommage (à définir, documenter et argumenter) par le candidat
- Qualité et lisibilité du code
- Le placement des tours est ergonomique et user friendly
- Gestion des différents types de projectiles

1.3 Planification initiale

La méthode AGILE a été utilisée afin de pouvoir s'adapté aux difficultés et aux changements d'architecture au cours du projet.

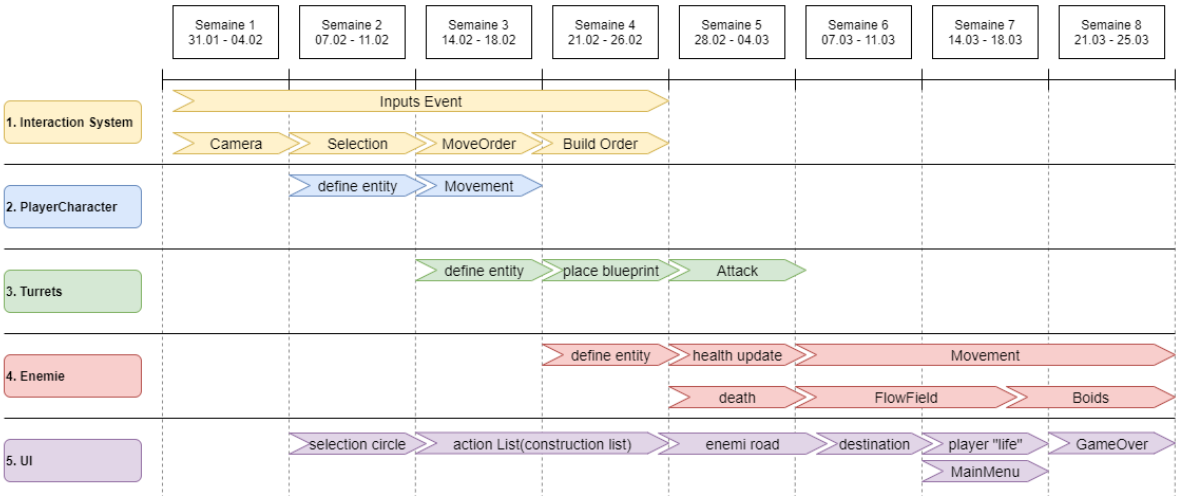
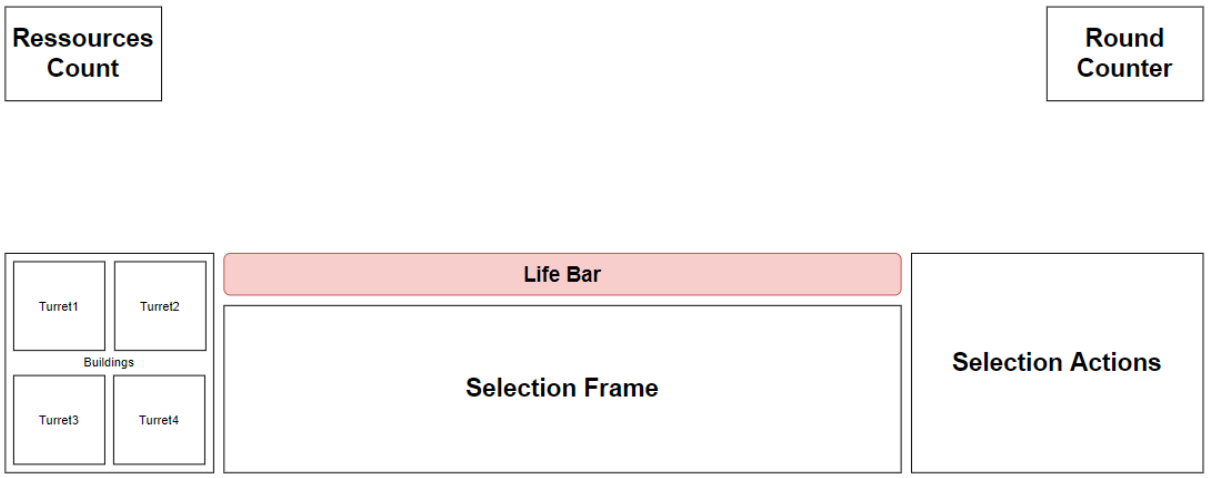


Figure 1: Planification Initiale

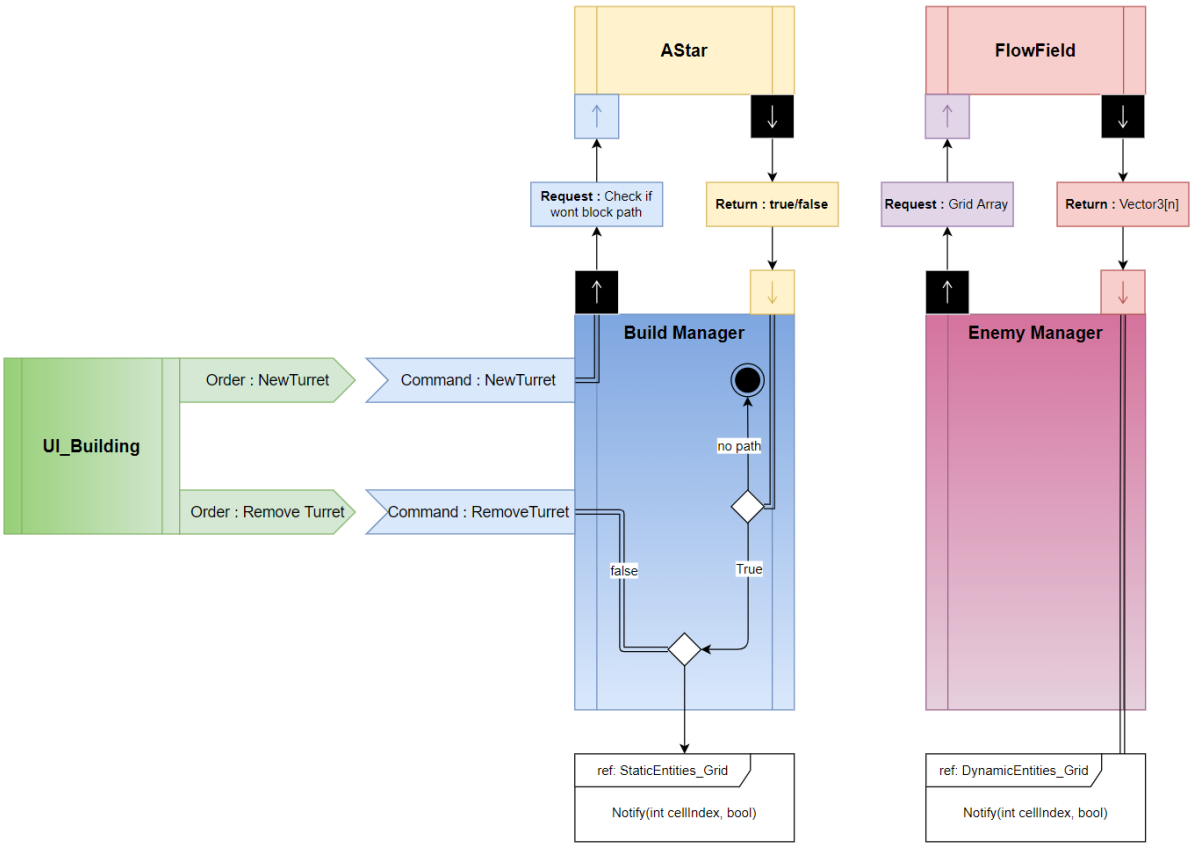
2 Analyse / Conception

2.1 Concept

2.1.1 Maquette de l'interface Utilisateur



2.1.2 Conception : Construction d'une tourelle :



2.2 Stratégie de test

Les tests se sont surtout concentrés sur le système de grille, étant l'élément central il était impératif que ce système soit solide, les tests ont par ailleurs révélé que les cas limites ont été la cause des bien des mots du projet, cependant ces cas limites n'étaient pas toujours évident à trouver et ont souvent été découvert au fur et à mesure des implémentations.

Action	Condition	Réaction
Partitionner la grille selon leur appartenance à un chunk	Taille : chunk = 2 ; cellule = 1 Taille : chunk = 4 ; cellule = 2	Chaque tranche de 4 éléments de l'array correspond aux cellules d'un chunk
Obtenir l'index de la cellule par rapport à la partition dont elle fait partie via le paramètre suivant : Index sur la grille	Taille : chunk = 2 ; cellule = 1 Taille : chunk = 4 ; cellule = 2	L'index retourné est le même que celui enregistré dans le dictionaire
Obtenir l'index de la cellule par rapport à la grille via les paramètres suivants : Index Chunk ; index de la cellule DANS le chunk	Taille : chunk = 2 ; cellule = 1 Taille : chunk = 4 ; cellule = 2	L'index retourné correspond à celui de l'array

2.3 *Risques techniques*

- **Performance de Unity / limitation de la machine du client**
  Je l'aborderai dans le document mais Unity est en pleine transition au niveau de son fonctionnement en général de ce fait il y a eu une stagnation a bien des niveaux en termes d'évolution de ce fait il y a des fonctionnalités qui ont très mal vieillies et qui m'ont demandé de redoubler d'effort et de recherches afin de pouvoir faire tourner le jeu sur la machine du client.
  (PS les principaux problèmes seront : l'interface utilisateur, l'animation de personnage)

  Solution : Si des soucis de performances apparaissent, utiliser les outils appropriés (Multithreading, Burst Compiler, éviter les « boxing » de mémoire).
- **Pathfinding / Recherche de Chemin :**
  Biens que des solutions génériques existent, il n'y a pas de solutions « génériques » à tout cas de figure, surtout en ce qui concerne l'adaptation à un terrain dynamique, il y a des nombreux articles de recherches qui traitent le sujet et propose des alternatives mais toutes pointent le fait que chaque solution à un compromis et que ce dernier doit être pris en compte afin de vérifier la compatibilité avec les exigences du projet et ses besoins.
  Il est aussi à noter que les solutions sont loin d'être simple dans leur compréhension et leur implémentation.

  Solution : Utilisation des méthodes dans leur forme simple et réserver leur forme complexe si le temps le permet
- **Mouvement de groupe des entités**
  Similaire au pathfinding, le mouvement d'entités en groupe et un sujet qui a fait couler beaucoup d'encres et qui tout bien qu'ayant aussi une implémentation de base (Reynolds Boids crée en 1986). Cette solution a cependant des limitations qui sont intimement lié au type de pathfinding utilisé et surtout au comportement souhaité ; de ce fait il faudra lors de la conception prendre en compte le type de pathfinding utilisé et au comportement souhaité et adapter la solution en conséquence.
- **Architecture de Code**
  Le projet devra faire face à un nombre conséquent de dépendances et partages de ressources entre les différents systèmes, il sera alors important de mettre ne place une stratégie permettant de centraliser les ressources afin d'effectuer une distribution efficace et surtout avoir un système d'évènements pour notifier les changements aux systèmes directement concernés.

3 Réalisation

3.1 Dossier de réalisation

- *Unity Engine*
- *Unity 2021.2.17*

3.2 Semaine 2 : Refactor du Planning

La Semaine 2 devait voir la partie "sélection du joueur" du projet naître, cependant il fut rapidement clair qu'un important changement dans le planning et la conception du projet était nécessaire.

3.2.1 Feature Inutile : Problème dans la définition des acteurs

Un premier prototype de sélection pour le personnage mais quelque chose sonnait faux dans le rôle de cette fonction.

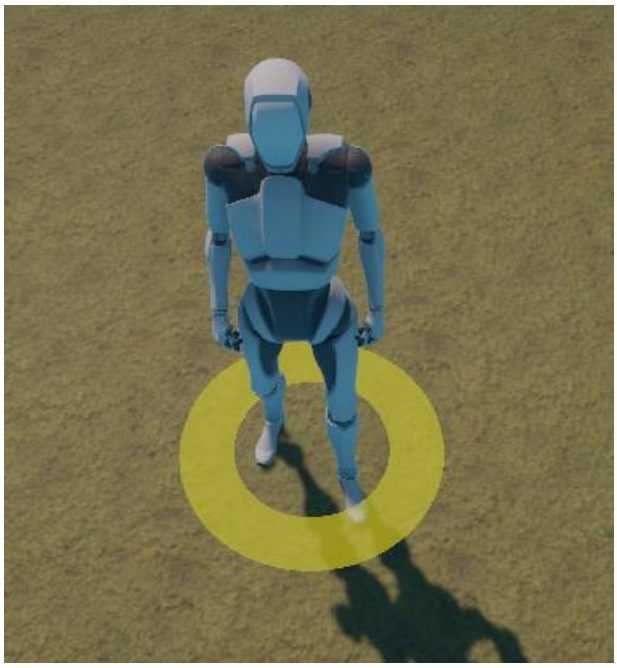


Figure 2 : Prototype de sélection

Quel objectif avait le personnage joueur mis à part voir visuellement un personnage avancer d'un point « A » à un point « B » ?

Eventuellement instaurer un délai entre les constructions, mais cela était-il absolument nécessaire et pertinent par rapport aux objectifs du cahier des charges ?

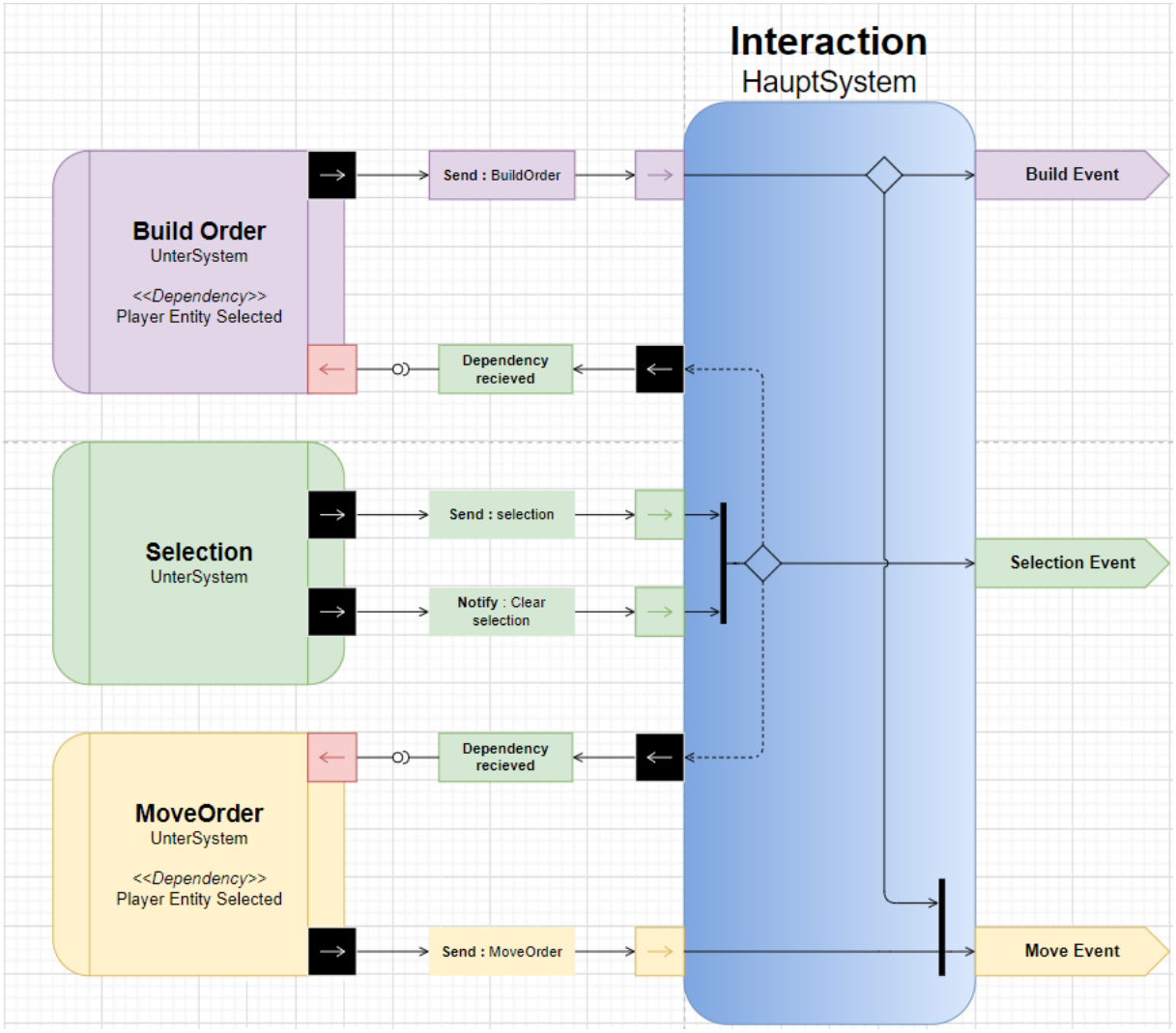


Figure 3 - Première Conception

Après réflexion il avérait que non seulement cet élément n'était pas utile au bon fonctionnement, mais de plus il ajoutait en complexité ; selon la première conception, des processus dépendent des actions du joueur pour être exécutés.

"MoveOrder" correspond au mouvement du joueur ; "MoveOrder" qui est lié à "BuildOrder" qui correspond à la construction de tour, la partie complète n'avait pas encore été finalisé mais il était prévu que la construction ne débute que lorsque le personnage atteint le lieu de construction.

3.2.2 Une Conception trop détaillé

Une erreur visible dans le schéma ci-dessus (voir : Figure 3 - Première Conception), une erreur commise à de nombreuse reprise dans plusieurs projets ayant entraîné leur abandon prématuré.
Une conception de base beaucoup trop détaillée, et la mise en place prématurée d'interfaces complexes (ci-dessus "HauptSystem" et "UnterSystem") donnant naissance à une énorme arborescence de dossier vide et des scripts qui ne seront

utilisées que bien plus tard et qui bien souvent, lorsque vient le moment de les implémenter; ces dernières ne correspondent plus à leur à la conception actuelle (liés à des refactors) et doivent être refactorisée voir complétement remodelées quand elles ne sont pas tout simplement retirées.

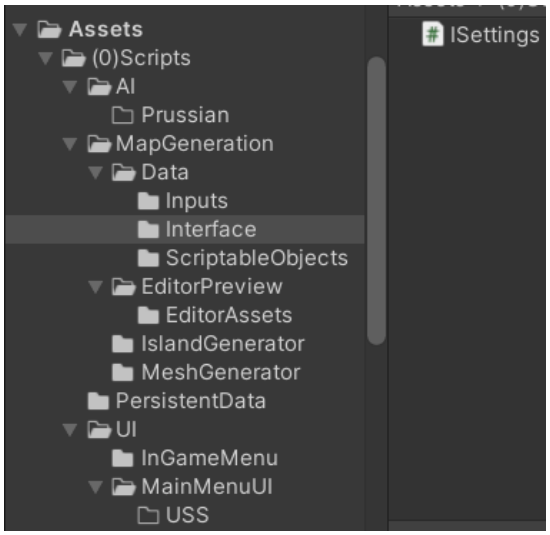


Figure 4 : Exemple d'un ancien projet abandonné

Exemple d'un projet abandonné illustrant ce qui allait probablement arriver.

Une arborescence qui sans être d'une grande complexité comporte des dossiers contenant un unique élément (Interface : ISettings) quand ils ne sont pas complétement vides (AI->Prussian, MainMenuUI->USS).

Ayant commis cette erreur par le passé je me suis documenté depuis le temps suivant des conseils via des interviews de développeurs sur Youtube ou encore en lisant des livres comme "The Pragmatic Programmer".
Un conseille revenait toujours qu'importe les sources.

> *When good-enough software is best, you can discipline yourself to write software that's good enough, good enough for your users, for future maintainers. (Hunt & Thomas)*

3.2.3 Conclusion après ces deux semaines de « perdu » :

Un général prussien à un jour dit :
« Aucun plan ne résiste au premier contact avec l'ennemi ». (Helmuth Von Moltke).

Je suis convaincu désormais que ce principe s'applique à la conception d'un projet quand elle est trop détaillée.

« *Aucune conception ne survie à son implémentation* »
(Probablement Mr. Pascal Hurni)

3.3 Tourelles :

3.3.1 Système de Tir : Première Solution (fonctionnelle)

La première implémentation naïve est un simple check autour de la tourelle afin de prendre pour cible le premier ennemi passant à portée de tir.

Cette solution est temporaire et est utilisée avant tout pour pouvoir tester les interactions de tir pour :

- La physique des munitions
- L'interaction avec les ennemies notamment pour la gestion de leur disparition
- Avoir un élément à présenter au Client

Cependant cette solution n'est pas viable dans le futur car elle à les faiblesses suivantes :

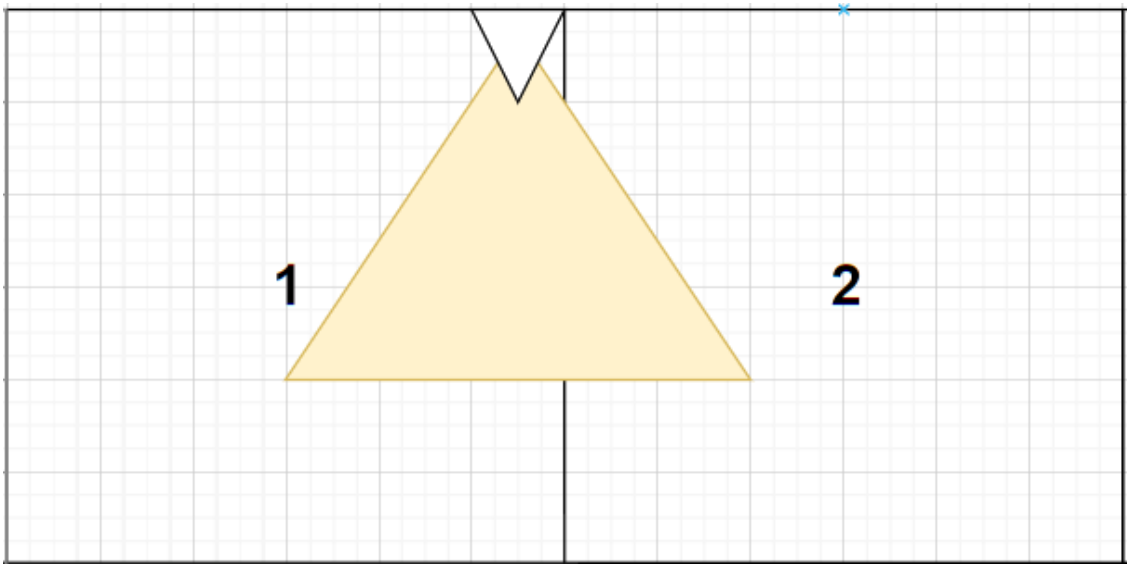
- La recherche de cible est effectuée pour chaque tourelle qui n'a pas de cible, ce qui veut dire que même si l'ennemi se trouve à l'autre bout du terrain la tourelle va effectuer sa recherche dans tous les cas (complexité O(n)).

- Les tourelles prennent pour cible le premier ennemi détecté qui bien souvent est le même, ce qui engendre une énorme concentration de tire sur une unique cible (le joueur risque de ne pas apprécier).

- La gestion des munitions n'est pas maintenable en l'état, j'étais parti sur un système de type Manager pour gérer les évènements de chaque munition ce qui a engendré beaucoup de complications et à complexifier le code, l'idée est de partir sur un système de pool d'objets.

3.3.2 Amélioration Future :

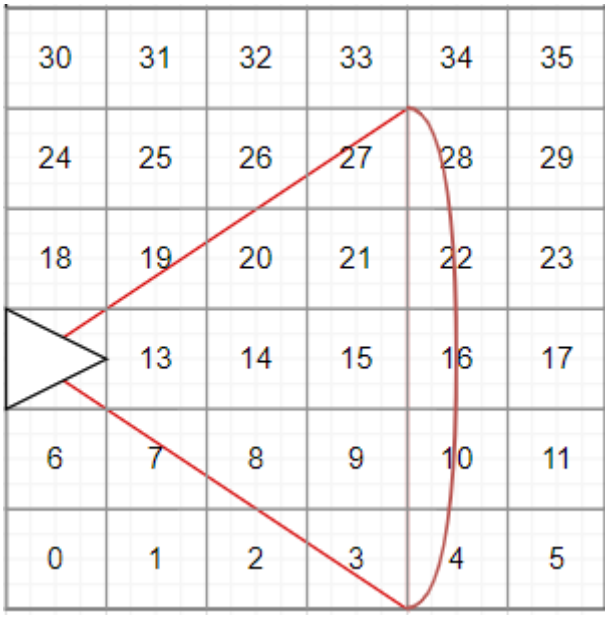
Activation de la recherche de cible par Chunk :
Chaque tourelle dépendant de sa portée activera sa recherche uniquement si un ennemi se trouve dans les chunks à portée de la tourelle



Concept 1 Ci-dessus : La tourelle sera active si une entité se trouve dans la partition 1 ou la partition 2

Ci-dessus : La tourelle sera active quand une entité se trouvera dans la partition 1 ou 2.

Visée en Cône
Pour le moment le système de visée utilise un cercle qui check les entités qui sont à dans son rayon.



L'idée de base était plutôt de partir sur une vue en cône.
Ci-contre une vue plus détaillée :

Solution plus élégante mais qui ajoute en complexité mais sera essentielle en terme de mécaniques de gameplay (développé plus loin)

3.4 Système de Grilles

Le projet Utilisant beaucoup de grilles, un utilitaire a été mis en place afin de facilité les différentes implémentations et surtout de les uniformiser afin d'éviter que chaque grille ne fonctionne d'une manière totalement différente.

Pourquoi cette utilitaire :

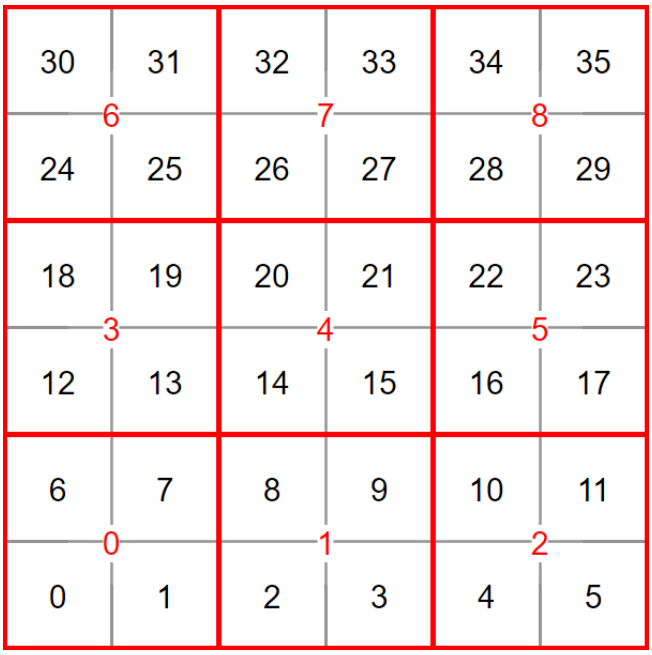
- Beaucoup de ressources communes à toute les grilles doivent être référencées à chaque nouvelle grille
- L'structure et les comportements sont fondamentalement similaires et doivent l'être pour les opérations de comparaison entre différentes grilles.
- Le grand nombre de référence croisée entre les grilles entraîne beaucoup de régression.
- Les opérations entres les grilles sont intrinsèquement lié à leur structure de base, tout changement/ajout de propriété ou extension doit être fait sur chaque grille

En Somme : Les grilles pour pouvoir fonctionner correctement entre elle doivent avoir suivre les mêmes règles, de plus il est nécessaire de standardiser les propriétés afin de d'avoir la même convention de nommage pour les méthodes, il y a jusqu'à 4 tailles différentes (voir convention de nommage) dans les propriétés il était donc important de fixer les noms afin d'éviter les erreurs liées à l'utilisation de la mauvaise propriété.

Les grilles suivantes prennent un type générique :

- GenericGrid<T> :
  - Une grille Simple contenant un Array Générique (T[N]).
- GenericChunkedGrid<T> :
  - Hérite de GenericGrid<T> et de ses propriétés.

Une grille partitionnée contenant un Dictionnaire (Dictionary<Int, T[ ]>),



Concept 2 : Grille Partitionnée
Dictionary [0] = {0, 1, 6, 7}

Note Concernant la grille partitionnée :
Cette dernière utilise le système de multithreading de Unity (Job System) et surtout le Compilateur Burst, concernant Burst, si le type générique est utilisé, ce dernier ne sera pas actif car au jour de l'écriture de cette documentation (01.03.2022), Burst ne peut pas être appelé via une méthode utilisant un type générique (cela provoquera un crash) le type doit donc être défini ou il faut alors utiliser la méthode sans burst.

3.4.1 Généralité

Convention de Nommage :
***Note Important** : la notation XY est utilisé pour faciliter la compréhension de la lecture à l'utilisation des méthodes ; cependant comme nous opérons dans un environnement 3D, l'axe Y correspondant désormais à la hauteur dans l'espace c'est donc les axes X et Z que nous souhaitons manipuler (correspondant aux axes directionnels du sol).*

- **MapSizeXY** = Taille XZ du terrain en jeu, la taille des cellules est toujours 1
- **NumCellXY** = Nombre de Cellules sur les axes XZ
  (MapSizeXY /taille Cellule)
- **NumChunkXY** = Nombre de partitions sur les axes XZ
  (MapSizeXY /taille partition)
- **NumCellInChunkXY** = Taille XZ du chunk par rapport à la taille des cellules
  (Taille Chunk / Taille Cellule)

Fonctionnalités :

- Souscrire des événements lorsque la grille subi une modification

- Modifier/obtenir un élément la grille
- Pouvoir comparer les valeurs de 2 grilles ayant des tailles des cellules différentes

3.4.2 Fonctionnalité de la grille partitionnée

La grille partitionnée apporte les complications suivantes :

- Les changements sur la partition (Dictionary) doivent aussi intervenir sur la grille (Array) et inversement.
- Il faut pouvoir retrouver les informations de la grille (Array) depuis la partition et inversement

Pourquoi ne pas avoir utilisé uniquement un dictionnaire ?
Dans beaucoup de cas il est plus facile de directement utiliser l'Array surtout dans le cadre d'interactions avec une grille simple et de réserver l'utilisation du dictionnaire aux fonctionnalités demandant une requête ciblant un chunk en particulier.

La conversion Array-Dictionnaire demande pour le moment beaucoup d'opération, ce qui peut impacter les performances.

3.4.3 Interactions entre les grilles :

Création d'un set d'interface pour gérer les interactions entre les grilles.
Le système répond aux besoins suivants :

- Accès à toutes les grilles du jeu.
- Centralisation des ressources communes nécessaires à la création des grilles
- Initialisation de toutes les grilles au démarrage
- Accès à la souscription des autres grilles

3.4.4 IGridHandler<T, GenericGrid<T>> :

Interface chargée de contenir les interactions de base nécessaire au bon fonctionnement de la grille, comprenant :

- Initialisation de la grille
- Accès au IGridSystem
- Souscription aux événements des autres grilles

3.4.5 IGridSystem :

Interface chargée de répertorier les grilles utilisées et lancer au démarrage du jeux les initialisations de chaque grille.

3.4.6 Problème : 2 Grilles – 2 Tailles de cellule

Lors de la création de l'utilitaire certaines problématiques sont apparu lié à l'état actuelle du projet

- Comment deux grilles avec des tailles de cellule différentes doivent-elles interagir ?

30	31	32	33	34	35
24	25	26	27	28	29
18	19	20	21	22	23
12	13	14	15	16	17
6	7	8	9	10	11
0	1	2	3	4	5

Figure 6 : Cellule Taille 1

6	7	8
3	4	5
0	1	2

Figure 5 : Cellule Taille 2

Ci-dessus :

- Si la grille 1 (taille 1) doit adapter ses informations par rapport à la grille 2 (taille 2) il devra modifier 4 cellules

- Si en revanche la grille 2 (taille 2) reçoit des informations de la grille 1 (taille1) ? est-il correct qu'une grande cellule soit entièrement modifiée par une cellule représentant ¼ de sa taille ?

Choix pour le projet :

- La grille des tourelles est de taille 2,
- La position des ennemis utilise une grille de taille 1.

Pour les besoins du jeu, le joueur ne doit pas pouvoir créer une tourelle quand une entité se trouve sur la cellule, donc si un ennemi se trouve dans la cellule correspondante (grille de la tourelle), la case entière est modifiée (même si l'ennemi ne couvre que ¼ de la cellule)

Comparaison de grilles ayant des tailles différentes

Diverses options ont été considérés chacune ayant leur compromis

Dans le cadre d'une comparaison type : Petite grille reçoit Grande grille

Lors de la comparaison :
Sur la petite grille, vérifier si la cellule concernée est comprise dans la grande grille

30	31	32	33	34	35
24	25	26	27	28	29
18	19	20	21	22	23
12	13	14	15	16	17
6	7	8	9	10	11
0	1	2	3	4	5

Figure 8 : GenericGrid<Int> Taille 1

true (index 6)	true (index 7)	true (index 8)
true (index 3)	true (index 4)	true (index 5)
false (index 0)	true (index 1)	true (index 2)

Figure 7 : GenericGrid<bool> Taille 2

Ci-dessus : la grille « Figure 6 » demande les informations de la grille « Figure 7 »

1) Calculer à chaque itération

Calculer si la cellule « taille 1 » est comprise dans la cellule « taille 2 ».
Dans l'exemple la cellule 7 est comprise dans la cellule index 0(false).

Avantage	Inconvénient
Information ciblée	Coût du calcul à chaque itération
Faible coût en mémoire RAM	Difficulté à prédire le nombre de comparaisons à faire
Efficient si peu de comparaison	Plus il y a de comparaisons à faire moins cette solution est efficace

Pire scénario : Grand nombre de comparaison (complexité O(n)).

2) Construire une grille ajustée à la taille du « demandeur »

Lors de la comparaison, une grille de la même taille que la plus petite sera construite sur la base des informations de la grande grille suivant le principe vu au point précédent (voir ci-dessus « calcule à chaque itération »).

Note : cette solution n'est envisagée que par le fait que le multithreading est possible La différence majeure entre la solution précédente est que le nombre d'itération est connu au départ et qu'aucune opération supplémentaire ne doit être effectué pour les obtenir.

Avantage	Inconvénient
Un calcul unique	Coût mémoire RAM
Efficient si le nombre de comparaisons à faire est grand	Gestion du multithreading (résultat)
Possibilité de multithreading	Non ciblé (Toute la grille est calculée)

Pire scénario : Peu de comparaison.

Choix pour le projet

La meilleure solution serait de pouvoir prédire le nombre d'itération et choisir entre les 2 méthodes en conséquence.
Par soucis de simplicité et du fait que Unity offre une solution de multithreading simple et « l'attribut magique » Burst pour booster les performances des calculs lors du multithreading ; la solution 2 est préférée.

Note Importante concernant le coût en mémoire

En dehors du coût lié au stockage permanent des grilles, les diverses opérations souvent doivent construire des arrays temporaires afin de stocker les résultats (surtout pour la grilles partitionnées) ce qui peut entraîner un grand nombre de boxing de valeur si ces opérations sont faites à la chaine pouvant entraîner un effet indésirable appelé « *les dents du dragon* » (bâtons rouges ci-dessous)

Figure 9 : Garbage Collector : les bâtons rouges représentent des allocations mémoires

Cet effet nous avertit que le Garbage Collector est beaucoup trop sollicité, impactant négativement les performances.

Solution :

Afin d'éviter ces allocations mémoires

- Les méthodes copieront directement les valeurs des NativeArray au lieu de passer par une conversion (en C# : Array.CopyTo)
- Lors du découpage des partitions, on utilise des NativeSlice (équivalent à ArraySegment<T>, Span<T> ou Memory<T> en C#) qui seront non pas de nouveaux arrays mais des références mémoires sur les éléments de l'array qui est « découpé », ces derniers seront ensuite passés en paramètre pour être directement copié dans la grille.

A noter que ces méthodes sont des overload, il est toujours possible d'utiliser des arrays au cas où l'utilisateur ne serait pas familier avec les containers cités ci-dessus ou pour faciliter l'implémentation de features tests.

3.5 Pathfinding / Recherche de Chemin

Pour le mouvement des troupes sur le terrain deux système ont été implémenter, chacun remplissant un rôle bien précis.

- FlowField : Grille de vecteurs donnant la direction pour les entités
- A*(ou aStar) : « Grille » permettant de savoir si le chemin n'est pas bloqué

Pourquoi deux systèmes ?
Chaque méthode de pathfinding (qui sont bien plus nombreuses) a son lot de compromis que j'expliquerais en détaille plus loin.

3.5.1 FlowField :

Avantage	Inconvénient
Une grille commune pour un groupe d'entité suivant le même objectifs	Coût mémoire RAM
Efficient si un grand nombre d'entité ont une destination commune	Beaucoup d'étapes de fabrication
Multithreading friendly	Difficile de déterminer si un chemin est bloqué

Ayant un grand nombre d'entité suivant le même objectifs, l'utilisation du FlowField allait de soi.

Cependant pour la création du FlowField il y a une grosse part d'artisanat, il y a une recette de base qui est certes fonctionnelle mais qui donne des résultats questionnables quant à l'esthétique.

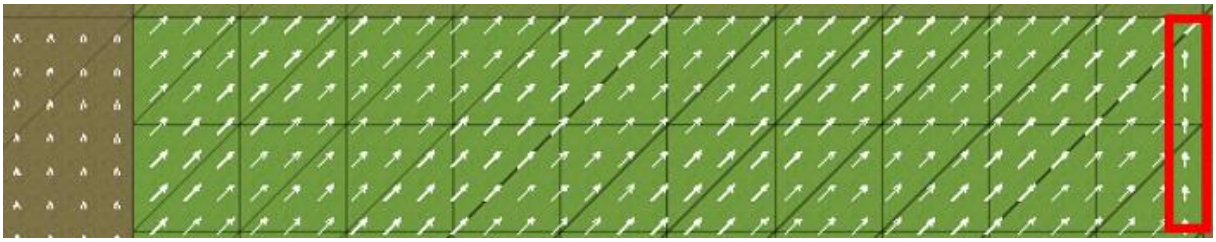


Figure 10: FlowField Implémentation basique

Ci-dessus (Figure 9) on voit le chemin prévu pour le passage des entités en vert, la partie brune représentant des zones « interdites aux piétons ». Le calcul cherchant le chemin le plus court va naturellement guider les vecteurs vers le côté le plus proche de l'objectif, hors dans notre cas (autant esthétiquement que mécaniquement nous préférerions que les entités marchent plutôt au centre de chemin.

Quelque chose dans ce genre-là :

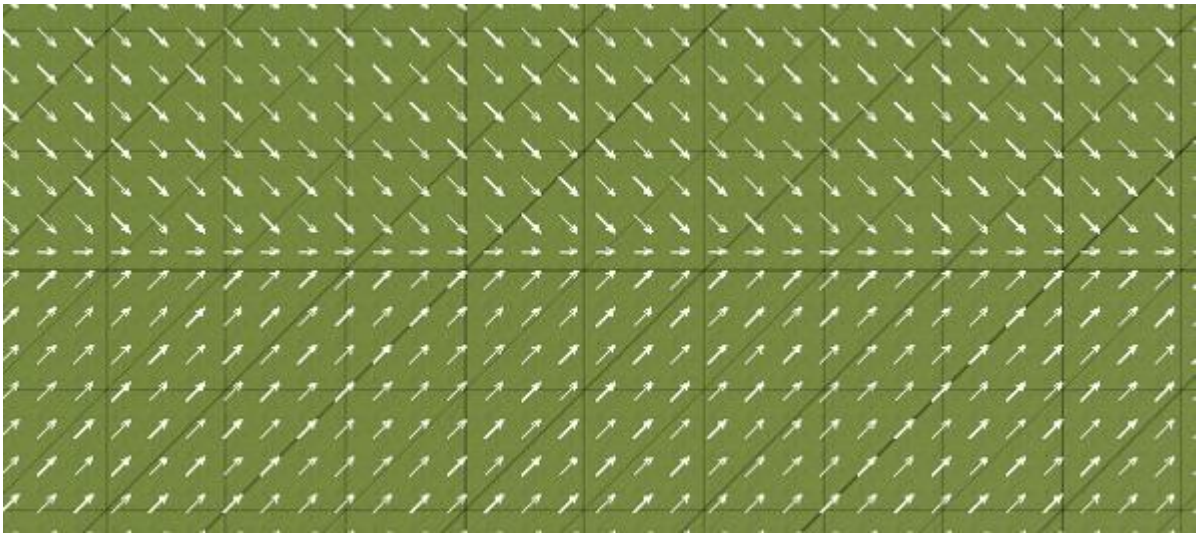


Figure 11: FlowField après ajustement

Pour atteindre ce résultat il a fallu jouer sur la grille des coûts des cellules (CostGrid) qui assigne une valeur qui va tromper l'entité en lui faisant croire que le temps pour aller d'une case à une autre est plus ou moins chère.
Pour l'Exemple ci-dessus la grille de coût a été ajuster comme suis :

- Plus la cellule est proche du bord, plus le coût est élevé.

Cet ajustement n'est qu'une des nombreuses possibilités de déformations qu'offre le FlowField.

Note : le système de chemin n'a pas survécu à l'implémentation qui a suivi.

3.5.1.1 Dans le Cadre du projet

Pour le projet les éléments suivants étaient à prendre en considération

- Utilisation d'une grille partitionnée
- Adaptation de la grille à de nouveaux obstacles
- Lissage des vecteurs
- Adaptation de la grille de coût aux menaces des tourelles

3.5.1.1.1 Utilisation d'une Grille Partitionnée

Pourquoi ?

L'utilisation d'une grille partitionnée permet de localiser les changements à effectuer, cependant ce point n'est possible qu'avec une implémentation complète d'un système HPA (Hierarchical Pathfinding) (voir documentation : Near Optimal Hierarchical Path-Finding).

Ce système prévoit l'implémentation de sous-objectifs, la notion de sous-objectifs à une grande importance car la dernière étape du FlowField se calcule par rapport à l'objectifs final et toutes les cellules allant jusqu'à ce dernier.
Cependant au stade du projet : Le temps ne permet pas de l'implémenter, il était plus judicieux de préparer le terrain pour le futur TPI.

3.5.1.1.2 Adaptation de la grille à de nouveaux obstacles

La grille doit pouvoir s'adapter aux nouveaux éléments placés par le joueur, il a fallu de ce fait ajouter la possibilité de recevoir une information extérieure. Ce qui posa deux problèmes :

1) Il fallait trouver une façon « élégante » mais surtout intuitive de plugger ces informations dans le l'objectif d'un maintien futur du projet.
2) Le FlowField fonctionne avec le job system ce qui implique qu'il est asynchrone. Il faut donc s'assurer que lorsque l'on reçoit l'ordre de recalculer le flowfield il faut d'abord terminer celui en court (s'il y en a un). Ce qui pose la considération suivante :
   Si plusieurs ordres arrivent dans un laps de temps court, ne serait-il pas judicieux de stocker ces ordres dans un buffer afin de les exécuter en même temps ? (Au lieu de forcer le job system à terminer les calculs en cours pouvant créer des freezes)

Concernant le premier point : Le temps à malheureusement manqué, de plus les implémentations qui découlaient de l'ajout des nouveaux obstacles n'étaient pas dans un état assez stable pour justifier de créer un système pour les faire communiquer proprement avec les grilles (Un important refactor serait nécessaire en premier lieu).

Pour le deuxième point : Le compilateur Burst même sur une machine modeste élimine le risque de freeze le temps de calcul étant inférieur à 0.001ms.

Il est à noter que la taille de terrain qui est assez modeste dans le cadre du projet aide beaucoup aux performances des calculs de grille. Les problèmes deviennent plus apparents à l'utilisation d'un terrain plus grand.

3.5.1.1.3 Mis au Banc

Les deux dernier points (Lissage des vecteurs et Adaptation de la grille aux menaces) ont malheureusement dû être sacrifié, faute de temps afin de prioriser des aspects plus essentiels au jeu, ces derniers éléments sont plus esthétiques que fonctionnels en termes de jeu.

3.5.2 AStar :

Avantage	Inconvénient
Facilité à déterminer si un chemin est bloqué	Coût en CPU proportionnel aux nombres d'entités
Très rapide	Multithreading compliqué
Peu d'étape de fabrication	Demande de stocker un chemin pour chaque entité

Le système A*(ou Astar) est un classique du pathfinding, il permet de trouver rapidement le chemin le plus cours d'un point A à un point B.

Ce système a été utilisé pour trouver rapidement si le un chemin est bloqué avant la pose d'une tourelle afin d'éviter que le joueur ne bloque les entités en bloquant l'accès à la porte de fin.

3.5.2.1 Dans le Cadre du projet

Pour le projet les éléments suivants étaient à prendre en considération

- Utilisation d'une grille simple
- Adaptation de la grille à de nouveaux obstacles
- Réduction du nombre d'opérations

3.5.2.1.1 Utilisation d'une Grille Simple

Le système Astar ne malheureusement n'a pas grand-chose à bénéficier d'une grille partitionnée en termes de performance du fait que tout est calculer séquentiellement et que chaque segment de l'algorithme dépend directement du résultat du précédent. De plus la complexité à adapter l'algorithme (déjà très rapide) afin que ce dernier bénéficie des atouts de la grille partitionnée n'est pas justifié en termes d'investissement de temps.

3.5.2.1.2 Adaptation de la grille à de nouveaux obstacles

Contrairement au flowfield ce système doit s'adapté quand les conditions suivantes sont réunies :

- On est en mode Construction
- La cellule dans laquelle se trouve la souris (position à l'écran) est sur le dernier chemin calculé

Contrairement au flowfield donc on calcul un chemin dans l'hypothèse d'un futur obstacle. Nous sommes donc dans l'hypothèse d'un futur bloqueur de chemin et si

cette hypothèse nous mène à une voie sans chemin, le système responsable de créer les tourelles va empêcher le joueur de poser la tourelle à l'endroit souhaitée.

3.5.2.1.3 Réduction du nombre d'opérations

Dans le cadre du projet, l'utilisation de l'algorithme n'étant que partielle, un certain nombre d'opération ont pu être retiré.

- Array.Reverse() : d'ordinaire le résultat à la fin du calcul est inversé, hors n'ayant besoin que de connaître son existence, cette étape est supprimée.
- D'ordinaire il est commun de supprimer le plus de « waypoints » en vérifiant si un obstacle se trouve entre 2 points.

3.6 Gestion des entités dynamiques (ennemies)

Synopsis : Cette partie a été la source de bien des maux et de tasses de café tant elle démontre que l'orienté objet n'est pas une solution adaptée à cette situation (contrairement à l'ECS).

Cette implémentation comprend les éléments suivants

- Mouvement des entités en utilisant le flowfield.
- Cache de toutes les entités présentent en jeu.
- Mise à jour du cache des entités

3.6.1 Update des entités

Pourquoi ne pas simplement utiliser la méthode Update implémentée par Monobehaviour sur chaque GameObject ?

Si la méthode Update est bien pratique elle est fortement déconseillée (voir blog Unity : https://blog.unity.com/technology/1k-update-calls), car à la manière dont fonctionne le framework de Unity, pour chaque objet ayant cette fonction, Unity va effectuer des sauts entre le script et le framework qui sont très coûteux.

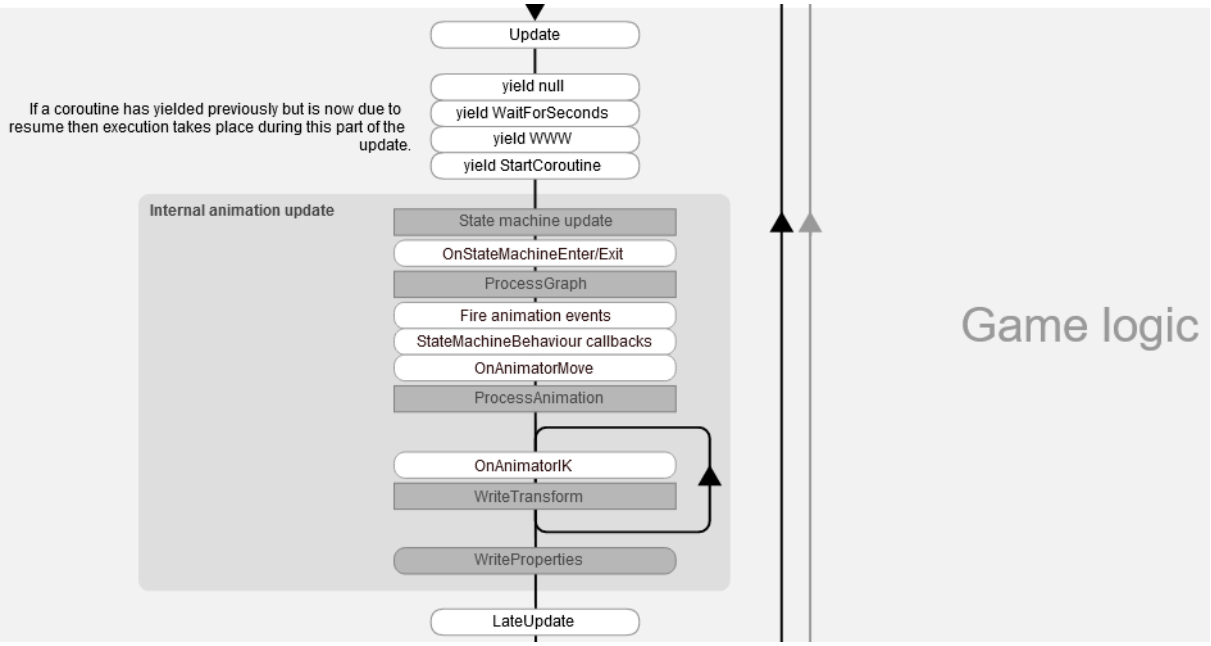


Figure 12: Partie GameLogic du cycle de vie de Unity (l'arbre entier est présent sur la documentation officielle)

3.6.1.1 Avantage de l'update Manager

L'update Manager en dehors des performances a aussi l'avantage de centraliser les actions faites sur tout un groupe à un seul endroit ainsi que les événements externes permettant un meilleur contrôle des événements influençant un groupe d'entités données.
Il a y de plus une possibilité de mettre à jour toutes les entités en parallèle via l'interface IJobParallelForTransform (Job System).

3.6.1.2 Gestion du buffer contenant les entités actives

Malgré le meilleur contrôle offert par l'update managers, les ressources n'en reste pas moins difficiles à coordonner, beaucoup de facteurs doivent être pris en compte :

1. **Maintien du Buffer contenant les entités**
   Cet élément est un central, il faut impérativement garder à jour le buffer au risque de faire crasher le programme qui tentera d'update des entités qui n'existe plus (le multithreading étant asynchrone n'ayant pas aidés sur ce point).
   La principale difficulté étant la gestion des événements extérieurs entrainant une modification du buffer (comme une tourelle détruisant une entité).

   Solution :
   Il a fallu s'assurer que la destruction de l'entité ne se fasse après que l'update (Update Manager) ait eu lieu et surtout pas avant. Unity Dans sa forme actuelle laisse peu de contrôle dans l'ordre d'exécution des scripts, il nous est donné « Update » et « LateUpdate » qui sont des updates sûres dans le sens

où elles seront exécutées à chaque frame (contrairement au FixeUpdate) et nous assure que LateUpdate sera exécuté après Update.

Ce faisant :

- Un buffer sous la forme d'un HashSet (pour éviter la gestion des doublons) informe qu'une entité doit être détruite.
- Lors du LateUpdate le Buffer contenant toutes les entités va se vider des entités contenues dans le HashSet (contenant les entités détruites)

2. L'accès aux ressources

L'Update étant effectué à chaque frame l'accès des entités est une question importante car l'opération étant faite à chaque frame il est important de prendre en considération la façon dont les ressources sont acquises et de rendre le plus directe possible l'accès aux ressources pour éviter les situations suivantes :

- GC allocation : Boxing the de valeurs liés à des callbacks retournant des arrays ou autres collections ; dans ce scénario l'allocation mémoire se fera à chaque frame créant l'effet « dents du dragon » abordé précédemment.
- Callback via des interfaces : Bien que donnant une meilleure lisibilité, l'abstraction augmente le cout en CPU pour chaque couche d'abstraction, si ces dernières sont peu nombreuses cela n'aura pas d'impacts cependant à mesure qu'elles augmentent, il sera impératif de garder un œil sur le profiler.

3.7 Erreurs restantes

Utilisation « cheap » du new Input System

Unity a depuis un moment revu la gestion des inputs dans unity introduisant un système permettant de facilement configurer différents contrôleurs (clavier, manette, etc..), épargnant aux développeur n*scripts pour chaque contrôleur que le jeu doit pouvoir recevoir et surtout il y a la possibilité de créer des templates réutilisables entre projet.
Par défaut Unity garde l'ancien système car ce dernier doit assurer la rétrocompatibilité.
L'avantage va plus loin que le « Cross-plateform », les features sont les suivantes :

- **Appel des inputs par Event ou via un Update**
  Le fait de pouvoir appeler les inputs via un système d'évènements est un changement massif car il permet de limiter les appels qui seraient à la fois rare et gourmant en calcul.
- **Les Inputs ont leur propre séquence dans la ligne de vie**

Le moment ou les inputs sont capturés n'ont jamais été très clair dans la documentation, cependant en regardant le profiler, il est possible de voir que les inputs sont les premiers à être mis à jour.

Le projet Utilise en ce moment majoritairement l'implémentation simple, à savoir mettre manuellement dans l'update une condition d'utilisation (clavier ou souris), un système similaire à l'ancien mais qui est toujours utilisable avec la nouvelle API.

Architecture non résiliente
La douloureuse expérience du refactor massif qu'a été l'ajour du système de grille a montré des faiblesses dans l'implémentation actuelle, la faute à beaucoup d'implémentation resté trop longtemps au stade « good enough », à des erreurs d'analyse ou de choix dans la conception de certains systèmes comme celui des balles (Update Manager au lieu d'un pool d'objets).
Si toutes ces implémentations ont permis de montrer un résultat au « client » elles ont, à force de les empiler créer une situation ou tout changement entraîne réaction en chaîne (problème de séparation des concernes).

Solutions envisagées :
- Réparer les problèmes de SOC avant toutes nouvelles implémentations
- Minimiser au maximum les dépendances entre les scripts

Le « Game Manager » est absent

Il n'y a pas encore de jeu à proprement dit, les mécaniques nécessaires au fonctionnement du jeu sont présentes mais il n'y a pas encore de système d'apparition d'ennemies ou encore de round.

Solutions :
Un system rapide peut-être implémenter sur le tas mais n'aurais que peu de valeur compte tenu de l'état actuelle du système d'interface qui est encore au stade de prototype.

Les fonctionnalités de l'interface utilisateurs

Bons nombres de fonctionnalités graphiques sont présentes et fonctionnel cependant leur implémentation demande un sérieux refactor, visible par la duplication de fonctionnalités similaire présente sur plusieurs éléments différents, quand d'autres comme le compteur de ressources ne sont qu'à moitié terminées.

Les actions du joueur sur les tourelles sont absentes

Toutes interactions prévues sur les tourelles omis leur placement sur la grille ont été coupé au montage fautes de temps, l'implémentation prématurée de cette fonctionnalité n'était pas judicieuse dans l'état des autres systèmes.

3.8 Liste des documents fournis

- *Article de recherches lus dans le cadre du projet*
- *Journal de travail*
- *Feuille de Résumer*
- *Cahier des charges*

4 Conclusions

4.1 Objectifs atteints / non-atteints

Non Atteints :

- Général :
  - Le temps de préparation n'a pas été implémenté
- Ennemies
  - Système de Round non implémenté
- Tourelle
  - Champs de vision en cône remplacé par une vision en cercle
  - Système de tir personnalisé en général a été repoussé car ce dernier dépendait du système de grille qui était un paramètre indispensable au bon fonctionnement du système de tir.
  - Le Système d'amélioration a aussi été coupé au montage, le programme ayant d'abord besoin d'un refactor dans son architecture avant d'ajouter un système aussi gros que ce dernier.

4.2 Points positifs / négatifs

Points Positifs :

- Le système de grille réutilisable pour les futurs projets notamment pour le TPI, bien que ce dernier n'est pas tout à fait complet dans ces fonctionnalités et mériterait un polissage, la base est assez solide pour être utiliser tel quel ajoutant un outil de plus à ma librairie personnelle.

- Améliorations dans l'implémentations de fonctionnalités, notamment la gestion des dépendances et de la séparation des concernes.

- Ajouts d'utilitaires/amélioration de ma bibliothèque personnelle.

Point Négatifs :

- Perte de temps au début qui aura coûté des fonctionnalités
- Incapacité à avoir vu la nécessité de créer un système de grille (pourtant indispensable) au début du projet
- Architecture encore non résiliente.
- Manque d'attention porté à la version « build » qui peut ne pas fonctionner comme sur la version éditeur.
- Éviter le « faux » travail, travailler sur des fonctionnalités décoratives qui bien que faisant partie d'une fonctionnalité prévue n'est pas nécessaire dans le bon fonctionnement du programme.
- Oubli de parler de certaines fonctionnalités importantes (comme le système de physique derrière les munitions) dans la documentation et ne s'en rendre compte qu'au moment où j'écris ces lignes (oups...).

4.3 Difficultés particulières

- Documentation WaterFall, au moment où j'écris ces lignes, cela fait 1 semaines que je passe à écrire cette documentation sans en voir le bout. Je pensais m'en sortir en écrivant des notes çà et là afin de ne pas oublier et aussi prendre des screenshots pendant le projet afin de m'éviter cette peine à l'écriture, mais l'écriture entière du projet est une expérience que je ne souhaite à personne, il faudra à l'avenir me forcer à prendre du temps pendant le projet pour écrire la documentation.

- Ne plus hésiter à développer une grosse feature dans un projet séparer, au départ de l'implémentation de la grille, j'ai fait la douloureuse expérience de tenter de créer l'utilitaire directement dans le projet, confrontant ainsi directement un outil aux règles déjà établi, ajoutant une complexité de plus à la conception d'un nouveau système, ce n'est qu'en fin de journée en faisant le bilan de l'avancement qu'il était clair que cette feature devait être

développer séparément afin d'éviter des corrections parfois inutile de par les nombreux changement qui sont opérés lors du prototypage.

- Trouver un moyen élégant de faire communiquer les différents managers, Le système actuelle bien que fonctionnel (Managers -> System) ne semble pas correcte et donne plus une impression de bricolage.

4.4 Possible Evolution

La principale implémentation qui se verra très probablement être amélioré/continuer est le système de grille, beaucoup de mes projets en ont utilisés et l'utilitaire créé durant ce projet est le résultat d'une année d'expérimentations.
L'utilitaire bien qu'utilisable n'est pas encore assez user-friendly à mon goût, il demande encore un certain nombre d'étapes et surtout d'écriture afin de pouvoir être utilisé.

Concernant le jeu en lui-même, comme mentionné dans « Erreurs Restantes » il ne manque qu'un élément pour que le jeu soit un jeu à proprement dit **cependant :**
Un aspect important de mon point de vue qui mériterait d'être revu est le game design ; les éléments ont été davantage fixés pour faire un projet que pour faire un jeu, le premier aspect à revoir serait à mon sens serait cette partie, car de ce dernier découle tout le « fun » du jeu qui est intrinsèquement lié au développement, car pourquoi créer un jeu si celui-ci n'a pas été pensé pour être amusant en premier lieu ?

5 Annexes

5.1 Résumé du rapport du TPI / version succincte de la documentation

5.2 Sources – Bibliographie

- Won-Ki Jeong and Ross Whitaker, A Fast Iterative Method for Eikonal Equations, March 23, 2007
- Michael L. Cui, Daniel D. Harabor, Alban Grastien; Compromise-free Pathfinding on a Navigation Mesh
- Xiao Cui and Hao Shi, A*-based Pathfinding in Modern Computer Games, IJCSNS International Journal of Computer Science and Network Security, VOL.11 No.1, January 2011
- Adrien Treuille1 Seth Cooper1 Zoran Popovi´c1,2, Continuum Crowds
- David Silver, Cooperative Pathfinding
- Marcel van der Heijden, Sander Bakkes, and Pieter Spronck, Dynamic Formations in Real-Time Strategy Games.

- Hao Li and Howard Hamilton, Flow Fields
- Adi Botea, Martin Müller, Jonathan Schaeffer; Near Optimal Hierarchical Path-Finding
- Harri Antikainen, Using the Hierarchical Pathfinding A* Algorithm in GIS to Find Paths through Rasters with Nonuniform Traversal Cost
- Ross Graham, Hugh McCabe and Stephen Sheridan, School of Informatics and Engineering, Institute of Technology, Blanchardstown; Pathfinding in Computer Games
- Michael Buro; Real-Time Strategy Games: A New AI Research Challenge

(Hunt & Thomas, p. 11) *The pragmatic programmer*

5.3 Table des Illustrations

Figure 1 : Prototype de sélection 10
Figure 2 - Première Conception..... 11
Figure 3 : Exemple d'un ancien projet abandonné 12