|  |
| --- |
| KaizerWald  Tower Defense |

Table des matières

[1 Analyse préliminaire 4](#_Toc98323549)

[1.1 Introduction 4](#_Toc98323550)

[1.2 Objectifs 4](#_Toc98323551)

[1.3 Planification initiale 5](#_Toc98323552)

[2 Analyse / Conception 5](#_Toc98323553)

[2.1 Concept 5](#_Toc98323554)

[2.2 Stratégie de test 5](#_Toc98323555)

[2.3 Risques techniques 5](#_Toc98323556)

[2.4 Planification 6](#_Toc98323557)

[2.5 Dossier de conception 7](#_Toc98323558)

[3 Réalisation 7](#_Toc98323559)

[3.1 Dossier de réalisation 7](#_Toc98323560)

[Semaine 2 : Refactor du Planning 8](#_Toc98323561)

[Une Conception trop détaillé 9](#_Toc98323562)

[3.2 Tourelles : 11](#_Toc98323563)

[3.2.1 Système de Tir : Première Solution (fonctionnelle) 11](#_Toc98323564)

[3.2.2 Amélioration Future : 12](#_Toc98323565)

[Solution plus élégante mais qui ajoute en complexité mais sera essentielle en terme de mécaniques de gameplay (développé plus loin) 12](#_Toc98323566)

[3.3 Système de Grilles 13](#_Toc98323567)

[3.3.1 Généralité 14](#_Toc98323568)

[3.3.2 Fonctionnalité de la grille partitionnée 15](#_Toc98323569)

[3.3.3 Interactions entre les grilles : 15](#_Toc98323570)

[3.3.4 IGridHandler<T, GenericGrid<T>> : 15](#_Toc98323571)

[3.3.5 IGridSystem : 15](#_Toc98323572)

[Placement des tourelles 19](#_Toc98323573)

[Pathfinding : FlowField 19](#_Toc98323574)

[Pathfinding : Ennemi 20](#_Toc98323575)

[3.4 Description des tests effectués 20](#_Toc98323576)

[3.5 Erreurs restantes 20](#_Toc98323577)

[3.6 Liste des documents fournis 20](#_Toc98323578)

[4 Conclusions 21](#_Toc98323579)

[5 Annexes 21](#_Toc98323580)

[5.1 Résumé du rapport du TPI / version succincte de la documentation 21](#_Toc98323581)

[5.2 Sources – Bibliographie 21](#_Toc98323582)

[5.3 Manuel d'Installation 21](#_Toc98323583)

[5.4 Manuel d'Utilisation 21](#_Toc98323584)

[5.5 Archives du projet 21](#_Toc98323585)

[5.6 Table des Illustrations 21](#_Toc98323586)

*NOTE L’INTENTION DES UTILISATEURS DE CE CANEVAS:  
Toutes les parties en italiques sont là pour aider à comprendre ce qu’il faut mettre dans cette partie du document. Elles n’ont donc aucune raison d’être dans le document final.*

*De plus, en fonction du type de projet, il est tout à fait possible que certains chapitres ou paragraphes n’aient aucun sens. Dans ce cas il est recommandé de les retirer du document pour éviter de l’alourdir inutilement.*

# Analyse préliminaire

## Introduction

*Ce chapitre décrit brièvement le projet, le cadre dans lequel il est réalisé, les raisons de ce choix et ce qu'il peut apporter à l'élève ou à l'école. Il n'est pas nécessaire de rentrer dans les détails (ceux-ci seront abordés plus loin) mais cela doit être aussi clair et complet que possible (idées de solutions). Ce chapitre contient également l'inventaire et la description des travaux qui auraient déjà été effectués pour ce projet.*

*Ces éléments peuvent être repris des spécifications de départ.*

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 je 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.

## Objectifs

*Ce chapitre énumère les objectifs du projet. L'atteinte ou non de ceux-ci devra pouvoir être contrôlée à la fin du projet. Les objectifs pourront éventuellement être revus après l'analyse.*

*Ces éléments peuvent être repris des spécifications de départ.*

## Planification initiale

*Ce chapitre montre la planification du projet. Celui-ci peut être découpé en tâches qui seront planifiées. Il s'agit de la première planification du projet, celle-ci devra être revue après l'analyse. Cette planification sera présentée sous la forme d'un diagramme.*

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

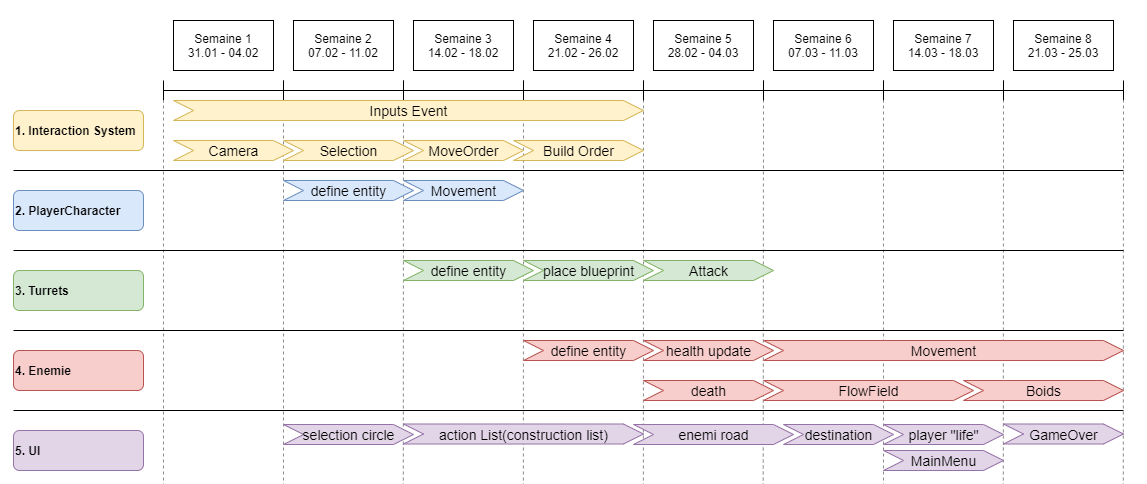


Figure 1: Planification Initiale

# Analyse / Conception

## Concept

*Le concept complet avec toutes ses annexes:*

*Par exemple :*

* *Multimédia: carte de site, maquettes papier, story board préliminaire, …*
* *Bases de données: interfaces graphiques, modèle conceptuel.*
* *Programmation: interfaces graphiques, maquettes, analyse fonctionnelle…*
* *…*

## Stratégie de test

*Décrire la stratégie globale de test:*

*Mettre :*

*Qu’est-ce que je teste*

*Les cas Limite*

*Resultat attendu*

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 |

## 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 et seulement 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.

## Planification

*Révision de la planification initiale du projet :*

* *planning indiquant les dates de début et de fin du projet ainsi que le découpage connu des diverses phases.*
* *partage des tâches en cas de travail à plusieurs.*

*Il s’agit en principe de la planification* ***définitive du projet****. Elle peut être ensuite affinée (découpage des tâches). Si les délais doivent être ensuite modifiés, le responsable de projet doit être avisé, et les raisons doivent être expliquées dans l’historique.*

## Dossier de conception

*Fournir tous les document de conception:*

* *le choix du matériel HW*
* *le choix des systèmes d'exploitation pour la réalisation et l'utilisation*
* *le choix des outils logiciels pour la réalisation et l'utilisation*
* *site web: réaliser les maquettes avec un logiciel, décrire toutes les animations sur papier, définir les mots-clés, choisir une formule d'hébergement, définir la méthode de mise à jour, …*
* *bases de données: décrire le modèle relationnel, le contenu détaillé des tables (caractéristiques de chaque champs) et les requêtes.*
* *programmation et scripts: organigramme, architecture du programme, découpage modulaire, entrées-sorties des modules, pseudo-code / structogramme…*

***Le dossier de conception devrait permettre de sous-traiter la réalisation du projet !***

# Réalisation

## Dossier de réalisation

* *les répertoires où le logiciel est installé*
* *la liste de tous les fichiers et une rapide description de leur contenu (des noms qui parlent !)*
* *les versions des systèmes d'exploitation et des outils logiciels*
* *Unity Engine*
* *Unity 2021.2.10*
* *programmation et scripts: librairies externes, dictionnaire des données, reconstruction du logiciel - cible à partir des sources.*

*Problème rencontré*

*Documentation pour la solution*

## 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.

### 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.

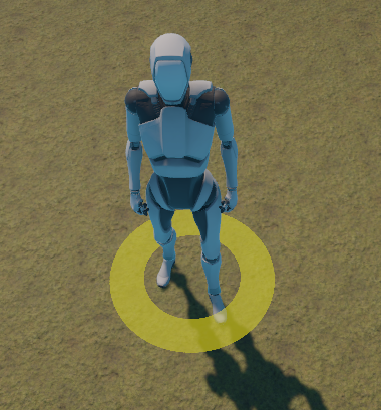


Figure 2 : Prototype de sélection

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 ?

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.

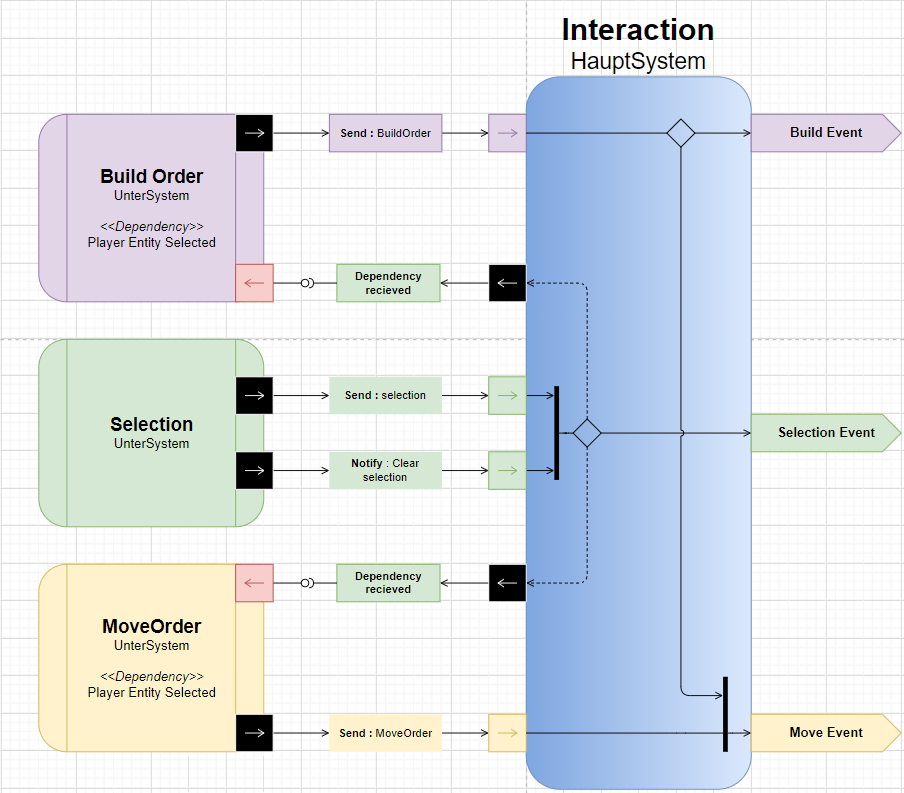


Figure 3 - Première Conception

*"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.*

### Une Conception trop détaillé

Une erreur visible dans le schéma ci-dessus (voir : Figure 2 - 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.

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

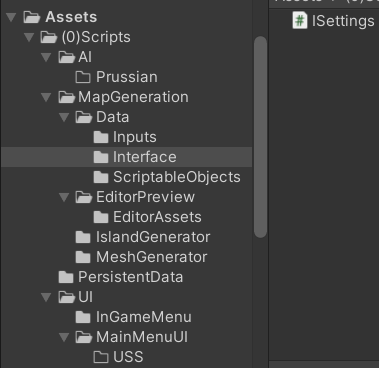


Figure 4 : Exemple d'un ancien projet abandonné

*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)

### 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)*

## Tourelles :

### 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

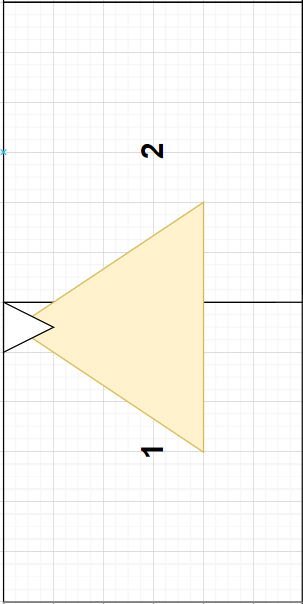
Cependant cette solution n’est pas viable dans le futur car elle à les faiblesses suivantes :

* 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.

### 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



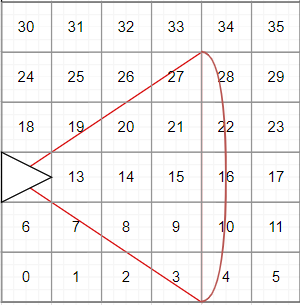
Concept 1 Ci-dessus : La tourelle sera active si une entité se trouve dans la partition 1 ou la partition 2

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)

## 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.

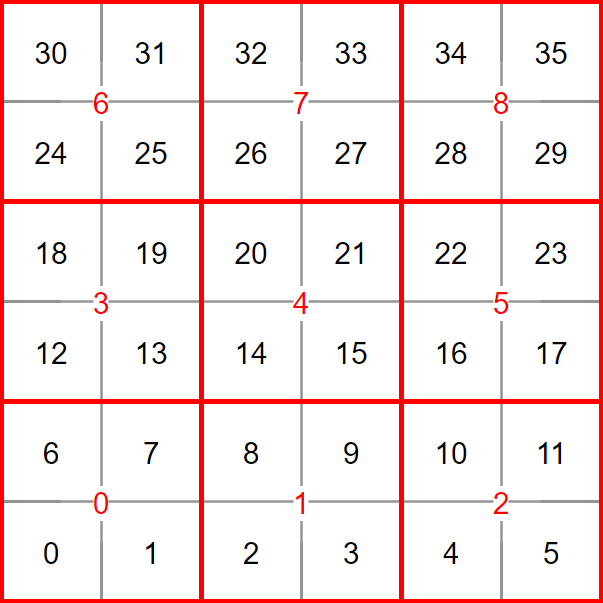
**Pourquoi cette utilitaire :**

* Beaucoup de ressources commune 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[ ]).
* 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.*

### 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 directionnelles 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

### 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.

### 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

### 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

### IGridSystem :

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

### 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 ?

Figure 5 : Cellule Taille 2

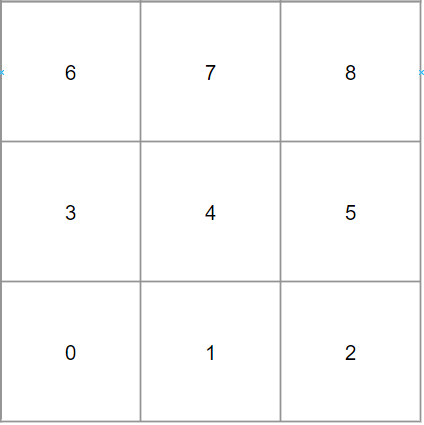
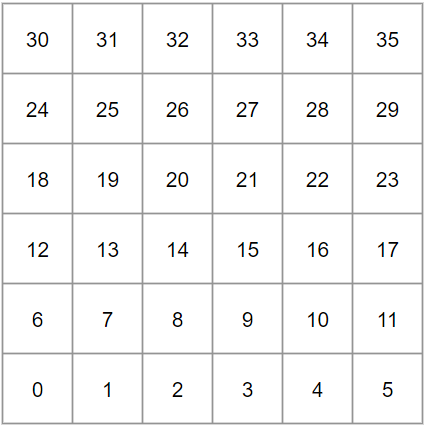


Figure 6 : Cellule Taille 1



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

Figure 7 : GenericGrid<bool> Taille 2

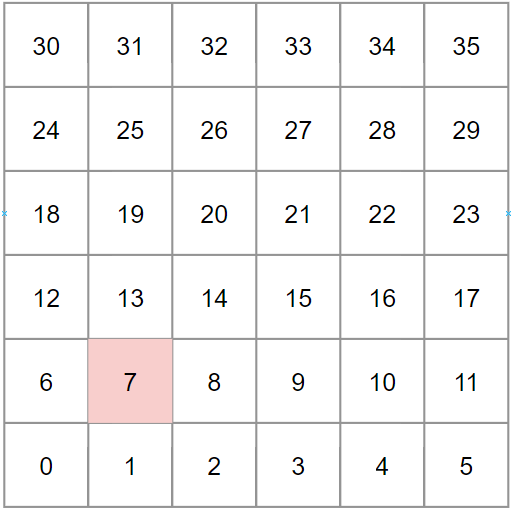
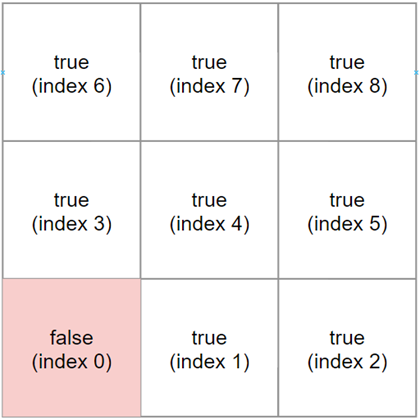


Figure 8 : GenericGrid<Int> Taille 1

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)).

1. **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 conséquences.

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)

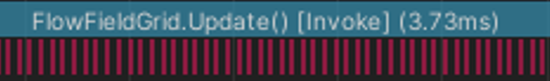


Figure 9 : Garbage Collector : les bâtons rouges représentent des allocations mémoires

Cette 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> ou Span<T> en C#) qui seront non pas un nouvel array mais une référence mémoire sur les éléments de l’array qu’ils découpent, 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.

## 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.

### 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 un 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.

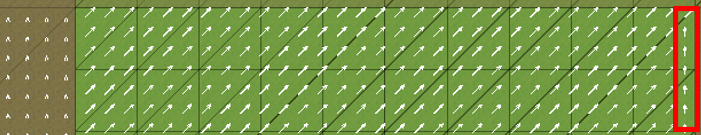


Figure 10: FlowField Implémentation basique

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 :

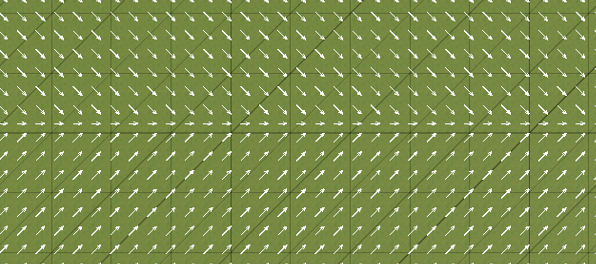


Figure 11: FlowField après ajustement

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.

Cette 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.*

#### **Dans le Cadre du projet**

Pour le projet les éléments suivant 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

##### **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 toute 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.

##### **Adaptation de la grille à de nouveaux obstacles**

La grille doit pouvoir s’adapter aux nouveaux éléments placé 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 au performances des calculs de grille. Les problèmes deviennent plus apparents à l’utilisation d’un terrain plus grand.

### AStar :

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

## Pathfinding : Ennemi

Option Pour le Pathfinding

Utilisation du FlowField pour le mouvement

Multithreading + Burst Compiler

Problème lié à l’animation dans unity au niveau des performances (DOTS ?)

NOTE : Evitez d’inclure les listings des sources, à moins que vous ne désiriez en expliquer une partie vous paraissant importante. Dans ce cas n’incluez que cette partie…

## Description des tests effectués

*Pour chaque partie testée de votre projet, il faut décrire:*

* *les conditions exactes de chaque test*
* *les preuves de test (papier ou fichier)*
* *tests sans preuve: fournir au moins une description*

## Erreurs restantes

*S'il reste encore des erreurs:*

* *Description détaillée*
* *Conséquences sur l'utilisation du produit*
* *Actions envisagées ou possibles*

## Liste des documents fournis

*Lister les documents fournis au client avec votre produit, en indiquant les numéros de versions*

* *le rapport de projet*
* *le manuel d'Installation (en annexe)*
* *le manuel d'Utilisation avec des exemples graphiques (en annexe)*
* *autres…*

# Conclusions

*Développez en tous cas les points suivants:*

* *Objectifs atteints / non-atteints*
* *Points positifs / négatifs*
* *Difficultés particulières*
* *Suites possibles pour le projet (évolutions & améliorations)*

# Annexes

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

## Sources – Bibliographie

*Liste des livres utilisés (Titre, auteur, date), des sites Internet (URL) consultés, des articles (Revue, date, titre, auteur)… Et de toutes les aides externes (noms)*

(Hunt & Thomas, p. 11)

## Manuel d'Installation

## Manuel d'Utilisation

## Archives du projet

*Media, … dans une fourre en plastique*

## Table des Illustrations

[Figure 1 : Prototype de sélection 8](file:///Z:\FPA\TPI\TowerDefenseDossierrDeProjet.docx#_Toc95580542)

[Figure 2 - Première Conception 9](file:///Z:\FPA\TPI\TowerDefenseDossierrDeProjet.docx#_Toc95580543)

[Figure 3 : Exemple d'un ancien projet abandonné 10](file:///Z:\FPA\TPI\TowerDefenseDossierrDeProjet.docx#_Toc95580544)