1. Motivations

N’étant pas un projet réalisé qui a pour but de résoudre un problème ou même juste qui a une certaine ‘utilité’, les motivations pour la création d’un jeu n’était que d’un point de vue académique. Le domaine du jeu vidéo est un domaine qui me passionne et mon superviseur à su voir l’intérêt de me laisser créer un jeu. N’ayant jamais eu 7la chance de créer un jeu vidéo dans le cadre d’un cours, je me suis dit que c’était un défi intéressant. Toute fois, je ne comptais pas faire un simple jeu qui existait déjà. Le jeu est donc un mélange de plusieurs éléments que j’aime beaucoup et qui rajoutait de la complexité à un type de jeu qui existe déjà. J’ai décider de combiner les styles tower defense et rôle play game. Le tower defense (souvent abrégée en TD) est un type de jeu vidéo où l’objectif est de défendre une zone contre des vagues successives d’ennemis se déplaçant suivant un itinéraire ou non, en construisant et en améliorant progressivement des tours défensives. Le role play game (souvent abrégée en RPG) est un type de jeu (vidéo ou non) où l’objectif est d’interpréter un personnage. Dans le domaine des jeux vidéo, il s’agit d’incarner et de ‘contrôler’ un personnage. Souvent le joueur a la possibilité d’améliorer son personnage et de le personnaliser. Dans le jeu 4Demons, le joueur a la chance de choisir un personnage qu’il contrôlera afin de l’aider dans sa quête de protection de base en plus d’utiliser un système de tour basique.

1. Approche de mise en œuvre
   1. Unity

Unity est un moteur de jeu multiplateforme développé par Unity Technologies. Un moteur de jeu est un ensemble de composants logiciels qui effectuent des calculs de géométrie et de physique utilisés dans les jeux vidéo. L'ensemble forme un simulateur en temps réel souple qui reproduit les caractéristiques des mondes imaginaires dans lesquels se déroulent les jeux. Le but visé par un moteur de jeu est de permettre à une équipe de développement de se concentrer sur le contenu et le déroulement du jeu plutôt que la résolution de problèmes informatiques. L’approche du logiciel Unity est orienté asset, un asset est une ressource informatique du jeu vidéo, par exemple, les images, les scripts, les niveaux, etc. Le script provenant de Unity qui contient toutes les fonctions des différents composants logiciels sont tous dans la classe MonoBehaviour. Pour utiliser ces fonctions, les nouvelles classes créer doivent héritées de MonoBehaviour. Pour qu’un script soit éxécuté, il faut que l’appel soit faire à partir d’un GameObject, qui est un object créer dans le jeu.

* + 1. Quelques fontcions principales de MonoBehaviour

La fonction Start() est appelé quand le script voulu est activé. C’est essentielle quand tu veux fixer tes paramètres avant que n’importe quelle autre fonction ne débute ou quand tu veux lancer des fonctions avant que la fonction Update() ne s’enclenche.

La fonction Update() est appelé à chaque image après que la fonction Start() ai été exécutée. Cette fonction dépend du temps, car si temps n’avance, les images par seconde tombe à 0, ce qui fait en sorte que la fonction Update() n’est pas appelé. Pour remédier à cela, il existe une alternative appelé FixedUpdate() qui est appelé en fonction du temps fixe, et donc qui s’écoule normalement.

Les fonctions OnEnable() et OnDisable() sont des fonctions qui permette l’activation et la désactivation de GameObjects.

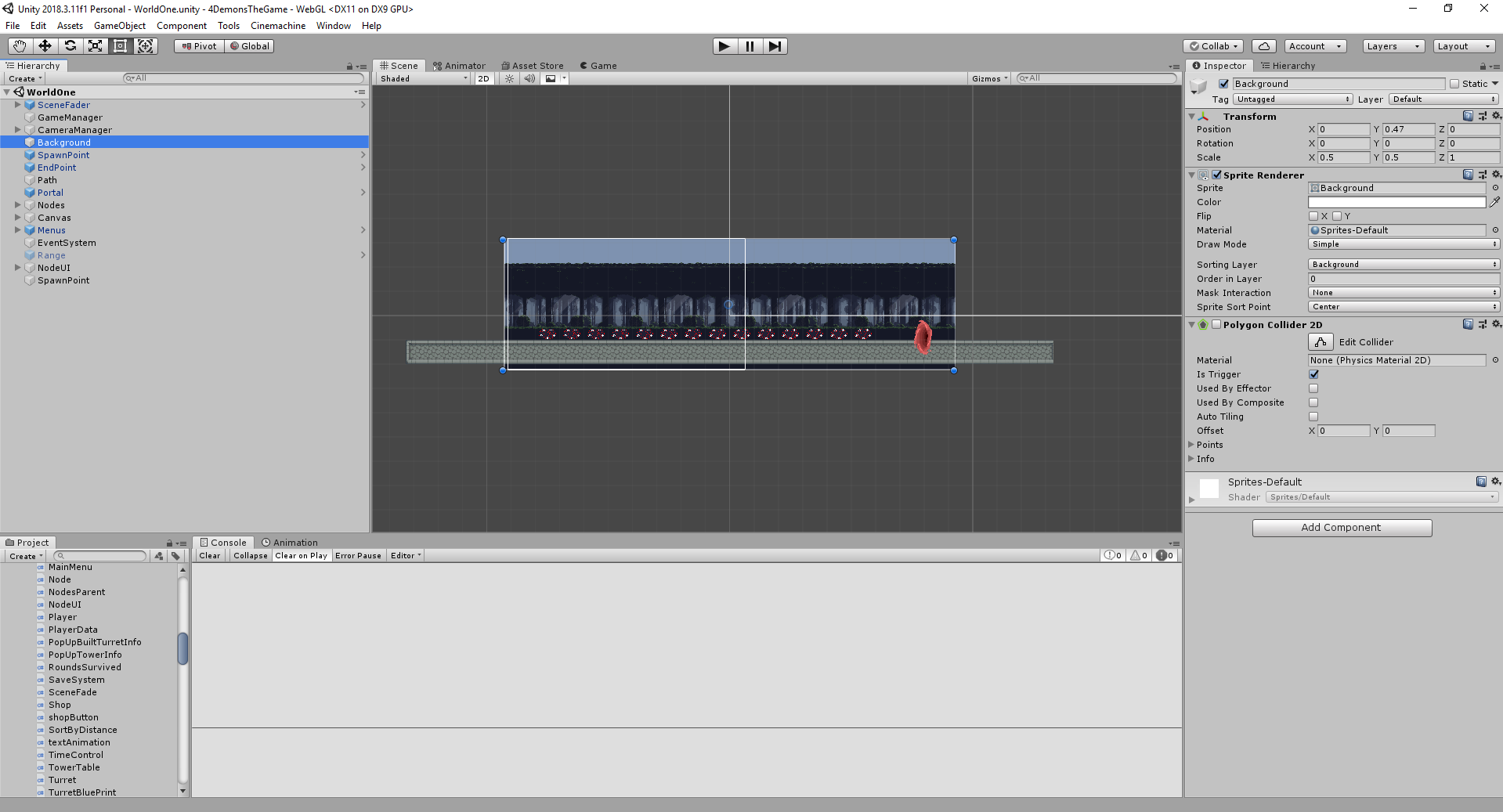


Figure : Unity UI

* 1. Architectures, composantes et interactions
     1. Les niveaux

Tous les niveaux commence avec des GameObjects de base : GameManager, SceneFader, CameraManager, Nodes, MenusAndUI, OnCalledMenus, NodeUI, Waypoints et GameSprites.

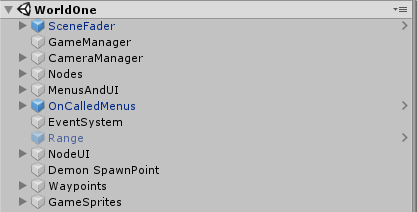


Figure : Level Hierarchy

* + - 1. GameManager

Le GameManager est l’unité centrale. C’est l’object qui contrôle majoritairement les actions du jeu et les états de celui-ci. Le GameManager a 4 scripts components.

Le premier est le **GameController** qui est responsable de gérer l’état du jeu.Je l’ai mis en Singleton, car je voulais qu’il n’y ai qu’un GameManager. Il sait donc quand la partie commence, quand le joueur a perdu et quand le joueur a gagner. Il gère aussi le temps, quand le jeu est en pause, en vitesse doublé ou en vitesse normal a l’aide du script TimeControl.

Le deuxième script est le **WaveSpaner** qui est le controlleur de l’apparition des ennemis et de la gestion des vagues. C’est dans le WaveSpawner qu’on conçoit la difficulté d’un niveau, qu’on ajoute le nombre et le type d’ennemis en fonction de la vague du niveau. C’est aussi a qu’on gère la vitesse d’apparition des ennemis et la temps entre chaque vague. \*insertion d’image du wave spawner\*

Le troisième script est le **BuildManager** qui est le controlleur de construction de tour. C’est aussi un Singleton, car je ne voulais qu’une instance de celui-ci. C’est dans ce script Que toute la magie de la construction de tour opère. Il contient toutes les tours, leur spécialité et leur caractéristique. Il y a aussi toutes les fonctions de constructions de tours. *Toute fois, je pense qu’un dictionnaire de tours à part (par exemple une classe nommée TowerDictionnary aurait été mieux et aurait augmenter la cohésion de cette classe.*

Pour finir il y a le script **LevelStatus**, qui lui contient toutes les informations pertinentes du niveau, soit l’argent de départ joueur, les points de vie de départ du portail. Il contient aussi, de manière statique, les vagues, ainsi que la vague actuelle, l’argent du joueur actuel ainsi que les points de vie du portail. J’ai mis ces attributs statiques pour qu’ils soit accessible de partout sans avoir à faire plein de référence vers LevelStatus. *En écrivant ce rapport, je me rend compte qu’une classe(script) LevelManager aurait été plus approprié pour la construction de nvieau, où on aurait pu avoir la construction des vagues et des paramètres initiaux du niveau.*

* + - 1. SceneFader

Comme le nom l’indique, l’objet SceneFader a comme script SceneFade, qui a 4 fonction.

La fonction Start() lance la coroutine FadeIn()(une coroutine est un type de fonction qui n’attend pas un return avant de laisser une autre fonction s’éxecuter, on peut donc appeler plusieurs coroutine qui s’execute selon le temps et selon un comportement donner).

La coroutine FadeIn() fait apparaître une image noir qui remplie l’ecran et après un certain temps diminue l’opacité pour devenir transparent pour faire un effet d’introduction.

La coroutine FadeOut() fait l’inverse, elle part d’une opacité nul à une opacité opaque. Toute fois elle prend comme argument un nom de Scene (les scenes continnent les environnements des différents niveau et menu d’un jeu) et permet de changer de scene à la scene voulu du string. *L’amélioration que je pourrais faire serait d’ajouter des try catch, car si un nom de scene est mal entrée, le jeu crash. Toute fois, pour l’ampleur de ce jeu, j’ai déjà testé manuellem7transitions, donc inutile, mais pour augmenter la réutilisation, ça serait une bonne option.*

* + - 1. CameraManager

Le CameraManager est le controlleur de la caméra. J’utilise un plugIn applelé Cinamachine qui offre des fonctions uniques de gestion de caméra. La fonction que j’utilise est Follow() qui suit un objet donnée en argument. Ici la caméra suit constamment le joueur.

* + - 1. Nodes

L’object Nodes contient tous les nœuds(nodes) du jeu. Ici ces nœuds sont les endroits où une tour peut être construite. Chaque object node on comme script compenent le script Node. Ce script fait aussi référence à plusieurs autres scripts : BuildManager (l’instance du singleton),GameController (instance du Singleton) Pop et Shop. Quand le joueur passe clique sur un node vide, le menu du Shop apparaît, il montre ensuite les 5 premières tours achetables. Si le joueur n’a pas assez d’argent pour une des tours, le boutons d’achats devient ‘’unclickable’’. La fonction qui vérifie l’argent est appelé dans le BuildManager. Si un joueur achete une tour, l’attribut turret (qui fait référence à la tour présente sur le node) prend comme GameObject la tour choisit par le joueur, le joueur est ensuite débiter et les valeurs Static d’argent du script LevelStatus est mis à jour. Une fois que la tour est acheter, le joueur peut faire 3 actions : l’améliorer/combiner la tour sur le node, vendre la tour ou simplement voir les caractéristiques de la tour.

Pour **améliorer/combiner** une tour, il suffit de cliquer sur la tour, un menu apparaitra pour montrer les combinaisons disponibles ainsi que l’amélioration possible. Les combinaisons sont généré au moment de la construction de la tour dans le script Node *(qui est une erreur à mon avis, un script combinaison aurait diminué le couplage et augmenter la cohésion de la classe)*. Une fois la tour combinée, le node est prend l’état ‘isCombined’, un booleen simple qui prend la valeur true *(encore une fois un patron d’état aurait été 100 fois mieux pour ce type de comportements)*. Pour l’amélioration de la tour instancié sur le node, la tour devient plus forte, avec un augmentation de ses dégats (*dans le futur changer la porté d’attaque, la vitesse d’attaque etc).* Le node est prend l’état ‘isUpgraded’, un booleen simple qui prend la valeur true.

Pour **vendre** une tour, le joueur clique sur le node le menu ouvre et il peut voir le bouton vendre. Quand le joueur vend une tour, le nœud est ‘’réinitialisé’’ il retourne aux états initiales, soit isCombined = false et isUpgrade = false, l’attribut turret devient null et on appele la fonction Destroy() (hérité de MonoBehaviour) qui détruit le game object dans la scene soit la tour qui était autre fois instanciée.

Pour **voir les caractéristiques** de la tour, le joueur ne fait que passer sa souris sur le node (où il y a une tour)(un appelle de fonction OnMouseHover() de la classe MonoBehaviour, le GameObject TowerOnFieldUi s’active. TowerOnFieldUi est un panneau indiquant les caractéristiques il a comme script component le script PopUpBuiltTurretInfo. PopUpBuiltTurretInfo a comme référence un target (un node), quand la souris passe sur un nœud, la cible du TowerOnFieldUi devient le nœud ciblé, il prend ensuite les valeurs de la tours pour mettre à jour les informations de son Ui avec les caractéstiques de la tour. Quand le joueur clique sur le nœud ou enlève la souris du nœud, le TowerOnFieldUi devient disabled et disparait en attendant sa prochaine réactivation sur le même nœud ou un nœud différent.

* + - 1. NodeUi

NodeUi est un GameObject qui a comme script NodeUI. Il contient 3 canvas pour 3 types de UI différent : ShopUi, ModifyTowersUi, TowerOnFieldUi. Le Script NodeUI gère absolument tout. Il contient toutes les références des zones de texte pour les différents Ui. Il gère aussi quel Ui est actif, ne voulant pas avoir plusieurs Ui actifs en même temps pour chaque node je dois vérifier si un Ui est actif ou non avant d’en faire apparaître un autre.

Le **ShopUi** est le Ui qui montre les tours achetables. De plus quand le joueur hover sur les tours, il peut voit les statistiques des tours pour bien choisir la tour qui lui convient. Ce n’est qu’un transfert de références.

Le **ModifyTowerUi**  est expliqué dans la section node, c’est la que le joueur voit les combinaison possible et les options de vente ou d’amélioration.

Le **TowerOnFieldUi**  est aussi expliqué dans la section node.

* + - 1. MenusAndUI

MenusAndUI est un Canva qui contient le bouton du Menu ainsi que le Indicator Panel un UI des statistiques du niveau. Dans Unity, les canvas sont les régions qui contiennent tous les UI.

Le bouton **Menu**  ouvre le menu qui se trouve dans OnCalledMenus.

Le **Indicator panel** est le UI où le joueur peut voir en temps réel sont avancés et sont status de jeu. Les valeurs des textsfields sont changer quand ils le doivent dans les différents scripts du jeu.

Le bouton FastForward est un bouton simple qui double la vitesse du temps quand il est activé via le script TimeControl et qui remet le temps à vitesse normale quand il est désactivé.

* + - 1. OnCalledMenus

OnCalledMenus contient 3 GameObjects : **MenuPanel, GameOver, CompleteLevel.** MenuPanel est le paneau du menu. Il est appelé quand le joueur appuie sur le bouton menu. Quand le menu apparaît le jeu se met en pause à l’aide de la fonction Pause() du GameController. Le menu a 6 boutons : **Return, Retry, Stats, Option, Levels, Quit.**

**Return** retourne au jeu, il ne fait donc rien autre que appelé la fonction Resume() de GameController qui remet le temps à la normale.

**Retry** reload la scene, ce qui remet le niveau à 0.

**Stats et Option** ne font rien pour l’instant, je ne voyais pas la complexité n’y l’utilité dans ses boutons et je n’ai pas eu le temps de les implémenter.

**Levels** envoie le joueur au menu de sélection des niveaux.

**Quit** envoie le joueur au menu principal.

Retry, Levels, Quit appellent la fonction GoToScene() du script MainMenu qui prend un le nom d’une Scene(string) en paramêtre. GoToScene() appèle ensuite la fonction FateTo du Scene Fader qui nous envoie à la scène demandé avec une belle animation.

* + - 1. Range

Range est un GameObject Simple dans le même sens que le TowerOnFieldUI (qui fait partie de l’objet NodeUI), il est invoqué de la même manière et est en fait un cercle qui représente la portée de la tour.

* + - 1. Demon SpawnPoint et Waypoints

Ce sont les points d’apparitions et de disparition des ennemis. Le Demon SpawnPoint est le point d’apparition du Demon que le joueur contrôle, il est fixe. Dans l’objet Waypoints, il y a le SpawnPoint et le EndPoint des ennemis. Ils sont utilisé dans le WaveSpawner et dans le script EnemyMovement. Les ennemis s’instancient à la même position que le SpawnPoint et il marche grace à la fonction Update() dans EnemyMovement. Quand l’ennemie est rendu au EndPoint, il meurt (Destroy) et le joueur perd un point de vie de sont portable.

* + - 1. GameSprites

GameSprites est seulement un Game Object qui contient des images soit le background et le chemin en pierre.

* + - 1. *Amélioration (Autres que les commentaires déjà mentionnés)*

*Il y a beaucoup d’amélioration que j’aurais aimer faire sur les niveaux. J’aurais aimé ajuster tous les scripts pour améliorer les fonctions pour respecter le Single Responsability Principle. Je vois dans mes scripts que je mets beaucoup de travail sur mes updates et ça devient parfois mélangeant. J’aurais aimé aussi ‘’patroniser’’ le tout. Je trouve que mon code et surtout mon projet aurait pu être mieux organisé si j’avais utilisé des patrons précis, comme des patrons de fabriques pour les types de tours ou des patrons d’états pour les nodes, le gamemanager, le timecontroller, etc.*

* + 1. Les tours
    2. Les ennemis
    3. Les démons
    4. Le joueur et le système de sauvegarde