UNIVERSITÉ DE MONTRÉAL

RAPPORT - 4 DEMONS

PAR

SAMI STEENHAUT

TRAVAIL PRÉSENTÉ À

A.S. HAFID

MICHALIS FAMELIS

TRAVAIL PRÉSENTÉ DANS LE CADRE DU COURS

PROJET INFORMATIQUE IFT-3150

PRÉSENTÉ LE 7 MAI 2019

Table des matières

[Motivations 4](#_Toc7896041)

[Approche de mise en œuvre 5](#_Toc7896042)

[Unity 5](#_Toc7896043)

[Quelques fonctions principales de MonoBehaviour 5](#_Toc7896044)

[Architectures, composantes et interactions 5](#_Toc7896045)

[Les niveaux 5](#_Toc7896046)

[GameManager 6](#_Toc7896047)

[SceneFader 6](#_Toc7896048)

[CameraManager 7](#_Toc7896049)

[Nodes 7](#_Toc7896050)

[NodeUi 8](#_Toc7896051)

[MenusAndUI 8](#_Toc7896052)

[OnCalledMenus 9](#_Toc7896053)

[Range 9](#_Toc7896054)

[Demon SpawnPoint et Waypoints 9](#_Toc7896055)

[GameSprites 10](#_Toc7896056)

[*Améliorations* 10](#_Toc7896057)

[Les tours 10](#_Toc7896058)

[Les ennemis 13](#_Toc7896059)

[Les démons 14](#_Toc7896060)

[Les composantes 14](#_Toc7896061)

[Amélioration 15](#_Toc7896062)

[Le joueur et le système de sauvegarde 15](#_Toc7896063)

[Le joueur(Player) 15](#_Toc7896064)

[PlayerData 15](#_Toc7896065)

[SaveSystem 15](#_Toc7896066)

[Améliorations 16](#_Toc7896067)

[Les résultats 17](#_Toc7896068)

[Les défis majeurs 18](#_Toc7896069)

[Unity 18](#_Toc7896070)

[Les vecteurs 18](#_Toc7896071)

[Les graphismes 18](#_Toc7896072)

[L’Itération 18](#_Toc7896073)

[Liens 19](#_Toc7896074)

[Annexe 20](#_Toc7896075)

# 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’étaient que d’un point de vue académique. Le domaine du jeu vidéo est un domaine qui me passionne et mon superviseur a su voir l’intérêt de me laisser créer un jeu. N’ayant jamais eu la 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 décidé 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.

# Approche de mise en œuvre

## 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 s'appelle MonoBehaviour. Il contient toutes les fonctions des différents composants logiciels fournies par Unity. Pour utiliser ces fonctions, les nouvelles classes créées doivent hériter de MonoBehaviour. Pour qu’un script soit exécuté, il faut que l’appel soit faire à partir d’un GameObject, qui est un Object créer dans le jeu.

### Quelques fonctions principales de MonoBehaviour

La fonction Start() est appelée quand le script voulu est activé. C’est essentiel 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 appelée à chaque image après que la fonction Start() ait 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 appelée. Pour remédier à cela, il existe une alternative appelée FixedUpdate() qui est appelé en fonction du temps fixe, et donc qui s’écoule normalement.

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

## Architectures, composantes et interactions

### Les niveaux

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

### GameManager

Le GameManager est l’unité centrale. C’est l’objet 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 ait qu’un GameManager. Il sait donc quand la partie commence, quand le joueur a perdu et quand le joueur a gagné. Il gère aussi le temps, quand le jeu est en pause, en vitesse doublée ou en vitesse normale à l’aide du script TimeControl.

Le deuxième script est le WaveSpawner qui est le contrôleur de l’apparition des ennemis et de la gestion des vagues. C’est dans le WaveSpawner que l’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 là qu’on gère la vitesse d’apparition des ennemis et le temps entre chaque vague.

Le troisième script est le BuildManager qui est le contrôleur 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 dans une base de donnée séparée (par exemple une classe nommée TowerDictionnary) aurait été mieux et aurait augmenté 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 soient accessibles de partout sans avoir à faire plein de référence vers LevelStatus. *En écrivant ce rapport, je me rends compte qu’une classe(script) LevelManager aurait été plus appropriée pour la construction de niveau, où on aurait pu avoir la construction des vagues et des paramètres initiaux du niveau.*

### SceneFader

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

La fonction Start() lance la coroutine[[1]](#footnote-1) FadeIn().

La fonction FadeTo() lance la coroutine FadeOut().

La coroutine FadeIn() fait apparaître une image noire qui remplit l’écran et après un certain temps diminue l’opacité pour devenir transparente pour faire un effet d’introduction.

La coroutine FadeOut() fait l’inverse, elle part d’une opacité nulle à une opacité opaque. Toute fois elle prend comme argument un nom de Scene (les scènes contiennent les environnements des différent niveau et menu d’un jeu) et permet de changer de scène à la scène voulue du string. *L’amélioration que je pourrais faire serait d’ajouter des try catch, car si un nom de scène est mal entré, le jeu crash. Toute fois, pour l’ampleur de ce jeu, j’ai déjà testé manuellement les transitions, donc inutiles, mais pour augmenter la réutilisation, ça serait une bonne option.*

### CameraManager

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

### Nodes

L’objet Nodes contient tous les nœuds(nodes) du jeu. Ici ces nœuds sont les endroits où une tour peut être construite. Chaque objet node a comme script component 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 bouton d’achats devient ‘’unclickable’’. La fonction qui vérifie l’argent est appelée dans le BuildManager. Si un joueur achète une tour, l’attribue turret (qui fait référence à la tour présente sur le node) prend comme GameObject la tour choisit par le joueur, le joueur est ensuite débité et les valeurs Static d’argent du script LevelStatus est mis à jour. Une fois que la tour est achetée, 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 apparaîtra pour montrer les combinaisons disponibles ainsi que l’amélioration possible. Les combinaisons sont générées 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 prend l’état ‘isCombined’, un booléen 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 instanciée sur le node, la tour devient plus forte, avec une augmentation de ses dégâts (*dans le futur, changer la portée d’attaque, la vitesse d’attaque, etc.).* Le node prend l’état ‘isUpgraded’, un booléen 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 appelé la fonction Destroy() (hérité de MonoBehaviour) qui détruit le GameObject dans la scène 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 de la tour posée. 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 tour 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.

### 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, 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 combinaisons possibles et les options de vente ou d’amélioration.

Le TowerOnFieldUi est aussi expliqué dans la section node.

### MenusAndUI

MenusAndUI est un Canva[[2]](#footnote-2) qui contient le bouton du Menu ainsi que le Indicator Panel un UI des statistiques du niveau.

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 textfields sont changées 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.

### OnCalledMenus

OnCalledMenus contient 3 GameObjects : MenuPanel, GameOver, CompleteLevel. MenuPanel est le panneau 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 qu’appelé la fonction Resume() de GameController qui remet le temps à la normale.

Retry reload la scène, 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 paramètre. GoToScene() appelé ensuite la fonction FateTo du Scene Fader qui nous envoie à la scène demandé avec une belle animation.

### 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.

### 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 utilisés dans le WaveSpawner et dans le script EnemyMovement. Les ennemis s’instancient à la même position que le SpawnPoint et il marche grâce à la fonction Update() dans EnemyMovement. Quand l’ennemi est rendu au EndPoint, il meurt (Destroy) et le joueur perd un point de vie de son portail.

### GameSprites

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

### Améliorations

*Il y a beaucoup d’amélioration que j’aurais aimé faire sur les niveaux. J’aurais aimé ajuster tous les scripts pour améliorer les fonctions pour respecter le Single Responsability Principle[[3]](#footnote-3). Je vois dans mes scripts que je mets beaucoup de travail sur mes updates et ça devient parfois mélangeant. Je trouve que mon code et surtout mon projet auraient pu être mieux organisés si j’avais utilisé des patrons précis, comme des patrons de fabriques[[4]](#footnote-4) pour les types de tours ou des patrons d’états[[5]](#footnote-5) pour les nodes, le gamemanager, le timecontroller, etc.*

### Les tours

#### L’architecture d’une tour

Chaque type de tour(15) est placé dans des fichiers à leur nom (ex. : FireTower). Dans chaque fichier il y a les 3 Prefabs[[6]](#footnote-6) qui composent une tour : Bullet, Effet, Tower.

#### Le GameObject Tower

La Tower est la base de la tour. C’est ce GameObject qui a comme composante le script Turret qui a pour but de contrôler le comportement de chaque tour. Une tour a un enfant : FirePoint qui est le point de départ d’une Bullet.Le script Turret contient tous les attributs d’une tour : la portée, la vitesse d’attaque, son degré d’amélioration, ses dommages de base, son coût. Il a aussi comme attributs le type d’ennemi que la tour doit prendre en cible, son type de bullet et une référence à sa cible.

La fonction update du script Turret appelle une fonction UpdateTarget() qui met dans une liste tous les ennemis instanciés et qui choisit le type de scan pour savoir quel ennemi sera pris en cible. Il y a 4 types de sélection : ShorthestEnemy(), FarthestEnemy(), FirstEnemy() et LastEnemy().

ShorthestEnemy() cible l’ennemi avec la distance la plus proche de la tour.

FarthestEnemy() cible l’ennemi avec la distance la plus longue de la tour.

FirstEnemy() cible l’ennemi le plus proche du EndPoint.

LastEnemy() cible l’ennemi le plus loin du EndPoint.

Chaque fonction fonctionne sensiblement de la même manière. Elle prend en paramètre un tableau de GameObject qui a comme tag[[7]](#footnote-7) le type d’ennemi voulu. Si le tableau n’est pas vide, un scan s’effectue autour de la tour à l’aide de la fonction Physics2D.OverlapCircleAll(point central du cercle, rayon) qui met tous les ennemis présents dans le cercle dans un tableau de Collider[[8]](#footnote-8). Par la suite je crée un dictionnaire avec tous ces GameObjects(ici les ennemis) avec comme Key la distance représentant la fonction (FirstEnemy => le plus proche du EndPoint) et le collider de l’ennemi comme l’objet associé à la clé. Je place ensuite les clés dans une liste simple et je la trie avec la fonction Sort() qui a un temps de complexité moyen cas de O(nlogn) et un temps et en pire cas O(n²). La complexité ici n’est pas vraiment importante, car il n’y aura jamais un nombre astronomique d’ennemis en même temps. Ensuite, en fonction du but de la fonction je sélectionne l’ennemi voulu dans le dictionnaire avec list.First() ou list.Last().

Quand la cible est choisie, le script appelle la fonction Shoot() avec comme intervalle le temps entre chaque attaque (la vitesse d’attaque).

Shoot() instancie une Bullet et appelle la fonction seek de cette bullet. Chaque tour instancie SON type de bullet.

#### Le GameObject Bullet

Le GameObject bullet a comme composante script AbstractBullet. Ici j’ai décidé de suivre le patron de conception Fabrique. Il y a un type de bullet pour chaque tour. Une AbstractBullet a comme attribut un rayon d’explosion, une vitesse, un effet et un type. Elle a aussi comme attribut une cible et un effet d’explosion.

La fonction Update est simple si elle la bullet n’a pas de cible ou si sa cible est morte, elle disparait. Elle se déplace de la même manière que les ennemis grâce à la fonction Vector2.MoveTo() héritée de MonoBehaviour.

Quand la balle rentre en collision avec un ennemi (les deux ayant des colliders) l’Effect d’explosion est instancié, et la fonction Hit() de AbstractBullet est appelée. Hit() prend en paramètre le dommage de la tour et la cible. La coroutine BulletSpecialDamage() est lancée et le collider de la balle est désactivé pour ne pas qu’elle Hit() plusieurs fois un ennemi ou d’autre ennemi, car elle ne peut pas être détruite tant que la coroutine roule. La coroutine BulletSpecialDamage() est une fonctionne abstraite. Pour chaque ConcreteBullet (ex. : FireBullet) qui hérite de AbstractBullet, la coroutine à un fonctionnement différent.

##### Algorithmes

Voici quelques algorithmes de la coroutine BulletSpecialDamage() en fonction du type de Bullet.

FireBullet : La bullet brule les ennemis. L’algorithme fait donc 1/10 du dommage toutes les 1/10 de seconde pendant ½ seconde. C’est une for loop qui tourne 5 fois en appelant la fonction TakeDamage() de la classe Enemy et entre chaque incrémentions la fonction attend 1/10 de seconde.

LightningBullet : La bullet paralyse l’ennemi pendant un certain temps. Elle réduit donc la vitesse de l’ennemi à 0, attend le temps de paralysie et puis remet la vitesse de l’ennemi à la normale.

EarthBullet : La bullet fait du dommage de zone. Elle a donc un rayon d’explosion, quand elle touche un ennemi, la fonction scan, de la même façon qu’une tour scan autour d’elle, pour voir les ennemis dans son rayon d’explosion et inflige des dommages à tous les ennemis dans son rayon. Elle prend aussi une fonction en paramètre qui peut être nul en cas de combinaison d’effet.

En effet, des balles peuvent avoir des combinaisons d’effet. Le procédé était plutôt simple, il suffisait de lancer deux coroutines en même temps, attendre que les deux soient exécutés, puis détruire la bullet.

#### Le GameObject Effect

L’Effect est l’effet que la balle fait quand elle explose. Il s’agit d’un particle system[[9]](#footnote-9) une composante de Unity qui permet de créer des effets sans programmation.

#### Améliorations

*Encore une fois, beaucoup d’amélioration me saute aux yeux. J’aurais aimé unifier un peu plus les bullets et le tour en ayant des types de tours de la même façon que les bullets ont des types. J’aurais aimé faire le même type de patron de conception pour les tours. Ceci implique que chaque prefab de tour aurait un script ConcreteTower dépendant du type de tour à la place d’avoir chaque le même script. De plus, il été mieux d’avoir comme enfant les bullets et les effets au prefab tour, encore une fois pour unifier une tour et pour forcer le fait qu’une balle de feu, peu seulement être à une tour de feu. De plus, j’aurais aimé ajouter dans les options, peut-être de chaque tour, ou bien toutes les tours à la fois, le fait de choisir le type de scan. Ça ajouterait de la stratégie, encore une fois par manque de temps je n’ai pas réussi à l’implémenter. On peut le changer dans l’inspecteur sur Unity, mais pas en jeu. Pour finir, j’aurais aimé améliorer les combinaisons d’attaques et ne pas seulement combiner les effets, mais en créer des nouveaux.*

### Les ennemis

Les GameObjects ennemis sont des prefabs. Il y a plusieurs types d’ennemis. Les ennemis ont comme composante un collider et deux scripts : Enemy et EnemyMovement.

#### Les composantes

EnemyMovement gère leur déplacement. Si un démon est sur le chemin de l’ennemi et qu’il n’a pas atteint sa capacité maximale de combat, l’ennemi va attaquer le démon, sinon il continue sa route vers le portail. Une fois que l'ennemi est arrivé au portail la fonction EndPath() s’occupe de réduire la vie du portail et de détruire l’ennemi. Si un démon est sur le chemin ou si le démon peut se faire attaquer(voir dans la section 2.2.4.) l’ennemi, va courir vers le démon.

Enemy gère le comportement de l’ennemi et ses caractéristiques. Un ennemi a une vitesse, une valeur (l’argent donné quand le joueur élimine l’ennemi), des points de vie, son dommage, sa portée d’attaque et une vitesse d’attaque. Il a aussi plusieurs status, isAfterPlayer pour savoir s’il court après le portail ou le démon, canAttack pour savoir s’il doit attaquer en fonction de sa vitesse d’attaque et sa distance jusqu’au EndPoint.

Il a comme fonction Update(), TakeDamage(), Attack() et Die().

Update() vérifie si l’ennemi attaque et s’il peut attaquer. Update() lance la coroutine Attack() quand la distance entre l’ennemi et le démon est plus petite que la portée d’attaque de l’ennemi.

La coroutine Attack() lance l’animation d’attaques de l’ennemi, puis appelle la fonction TakeDamage() du démon. Elle attend que l’animation soit terminée, puis repart le cooldown entre chaque attaque.

Die() détruit l’ennemie, lance l’effet de mort qui est un particle system, update le status du niveau et ajoute l’argent au joueur.

TakeDamage() est appelé quand une balle rentre en collision avec l’ennemi ou si le joueur attaque l’ennemi avec le démon. La fonction enlève de la vie à l’ennemi et vérifie si la vie de l’ennemi est plus petite ou égale à 0, si c’est le car, l’ennemi meurt.

#### Améliorations

*La modification du code en implémentant un patron de Fabrique est une amélioration essentielle du jeu qui permettrait la création d’ennemis différents et uniques. Chaque ennemi aurait leur façon de se déplacer et leur façon d’être. J’aurais aussi aimé avoir beaucoup plus d’ennemis, mais faute de temps il y en a que troie.*

### Les démons

En principe, il devrait y avoir plusieurs démons, mais faute de temps, seulement un démon a pu être créé. Le joueur contrôle un démon. Il peut marcher et attaquer les ennemis qui défilent sur le chemin. Si le démon meurt (si le joueur est éliminé), le joueur perd la partie.

#### Les composantes

En plus des composantes basiques d’un GameObject sur Unity, le démon a deux composantes scripts : Demon et DemonController.

DemonController gère le mouvement du démon. Le script contient la vitesse du personnage et son vecteur de direction. Il y a trois fonctions principales : ProcessInputs(), Move() et Animate(). ProcessInputs() et Move() sont appelées dans la fonction Update() héritée de la class MonoBehaviour.

ProcessInputs() utilise l’interface Input qui permet de lire les Conventional Game Input (ici les flèches du clavier). La fonction change la direction du vecteur en fonction des boutons appuyés.

Move() est appelé après ProcessInputs(), la fonction déplace le démon dans la scène en fonction du vecteur de direction.

Animate() gère l’animation. Si le personnage bouge donc si le vecteur de direction n’est pas nul, l’animation de course commence, sinon le démon idle.

Demon est le script qui gère les états du démon. Tous les attributs du démon sont là : sa vitesse d’attaque, sa portée d’attaque, ses dommages, ses points de vie, son armure, le nombre d’ennemis qu’il peut combattre à la fois et sa régénération de vie.

Attack() est une coroutine. Elle est appelée dans update chaque fois que la touche espace est appuyée. Il y a un point d’attaque devant le démon qui a une certaine portée(un cercle) quand le démon attaque, il inflige des dégâts à tous les ennemis dans la zone.

TakeDamage() est la fonction qui gère les dommages que le démon subit. Quand il se fait attaquer par les ennemis, ses points de vie diminuent. Les dommages subis représentent les dommages de l’attaque des ennemis moins l’armure du démon.

Regeneration() est la fonction qui gère la régénération de vie. Elle est appelée dans Update(). Selon les attributs du démon, il récupère se montant chaque 0.5 seconde.

Une fois que le démon atteint 0 point de vie, son status change à isDead = true, quand c’est le cas le GameController lance la procédure GameOver.

#### Amélioration

*Du même type que les autres améliorations, j’aurais adoré construire les démons en suivant un patron de fabrique. Ceci me permettrait de créer différents démons avec leur propre script. J’aurais aimé avoir un système d’état suivant un patron d’état qui aurait géré le status du démon et qui me permettrait d’en ajouter plus facilement par exemple : point de vie bas, mort, endormi, brulé, etc.*

### Le joueur et le système de sauvegarde

#### Le joueur(Player)

Player est un singleton qui prend en compte tous les états et les avancements du joueur. Il a 5 attributs : le niveau atteint, le nombre de TimeLessSouls[[10]](#footnote-10), le fichier de sauvegarde actuel, le nombre d’ennemis qu’il a tué et le démon qu’il jouera.. Il y a deux fonctions importantes dans Player.

Save() qui s’occupe de sauvegarder l’avancement du joueur (ses attributs). Elle est appelée toutes les cinq secondes.

Load() qui permet de charger les attributs de la sauvegarde choisie dans le joueur. Elle est appelée quand le joueur choisit sa sauvegarde au début du jeu.

#### PlayerData

PlayerData est une classe sérialisable[[11]](#footnote-11). Elle sert seulement à contenir de l’information sur le joueur. C’est cette classe qui se situe entre le système de sauvegarde et la classe Player.

#### SaveSystem

SaveSystem est la classe où toutes les fonctions de créations et de chargement de sauvegarde opèrent.

SavePlayer() prend un joueur en argument et un fichier de sauvegarde. Elle sauvegarde les données du joueur en binaire. On utilise un formateur BinayFormatter qui format les données dans le fichier fournit en argument. C’est aussi cette fonction qui créera les fichiers de sauvegarde. Elle est appelée quand Player l’appelle et quand le joueur créera une nouvelle partie.

LoadPlayer() prend un fichier de sauvegarde en argument et retourne un PlayerData. LoadPlayer() va chercher les informations du fichier choisi et utilise le formateur binaire pour désérialiser l’information. Il retourne ensuite le PlayerData qui était l’information sérialisée dans le fichier de sauvegarde. Si le fichier de sauvegarde n’existe pas, dans notre cas s’est en cas d’erreur(préventif) ou bien en cas de création de nouvelles parties, LoadPlayer() appelé la fonction SavePlayer avec comme argument le fichier vide(non créé) et un nouveau Player(new Player). Ceci crée un nouveau fichier de sauvegarde. Ensuite il charge le PlayerData de la nouvelle partie.

#### 

#### Améliorations

*Il y a énormément d’amélioration possible pour ce qui est de Player. Premièrement, toutes les informations sur les options devraient faire partie du PlayerData. Deuxièmement l’ajout de d’autre attribut serait très intéressant comme une base de données des démons utilisés qui permettrait d’ajouter de la personnalisation et de l’amélioration aux démons en fonction du joueur. On pourrait aussi ajouter beaucoup plus de statistiques sur le joueur. Pour ce qui est du système de sauvegarde, j’aurais aimé ajouter une fonction CreateSave() qui aurait permis la création de sauvegarde ainsi que la fonction DeleteSave() qui permettrait la suppression.*

# Les résultats

Je suis très satisfait des résultats! Le jeu est jouable, les bases sont là. Certes il y a beaucoup beaucoup d’amélioration possible, mais pour un projet de 3-4 mois, je trouve que les résultats sont très bien. Le jeu est jouable sur le site : <https://www.webdepot.umontreal.ca/Usagers/p1132449/MonDepotPublic/IFT3150/4demons.html?uniq=-havazl>.

# Les défis majeurs

Les défis majeurs se regroupent dans un grand ensemble qui est le manque d’expérience.

## Unity

J’ai eu beaucoup de misère avec la gestion des classes, n’ayant jamais travaillé avec un moteur (Engine) l’organisation des classes était parfois, même souvent, assez abstraite, car beaucoup d’éléments étaient gérés par le moteur à l’aide de la classe : Monobehaviour.

## Les vecteurs

Je n’ai pas réalisé à quel point les vecteurs étaient importants dans le monde du jeu vidéo. J’ai eu énormément de difficulté à voir comment bien utiliser les vecteurs pour me permettre de réaliser certaines actions.

## Les graphismes

Ceci n’est pas relié au code, mais l’ajout de graphisme et d’animation a été très difficile et a pris beaucoup de temps.

## L’Itération

Avec le manque d’expérience vient l’acquisition de connaissances. Autant ceci est une bonne chose, autant dans un projet qui doit être remis à un temps fixe ce l’est moins. Je dis cela, car plus j’avançais dans le développement du jeu, plus je voulais changer certaines mécaniques du jeu. Par exemple, ce n’est qu’à la fin de cette session que j’ai réellement compris l’utilisation de patrons de conception. Ceci m’aurait permis d’avoir beaucoup plus de contenus et d’avoir une meilleure structure du projet. C’est en ayant le cours IFT 3911 : Analyse et Conception des Logiciels, avec Eugene Syriani, que j’ai pu comprendre énormément les comportements du logiciel.

# Liens

Lien GitHub : <https://github.com/KoldLab/4Demons>

Lien du site : <https://www.webdepot.umontreal.ca/Usagers/p1132449/MonDepotPublic/IFT3150/4demons.html?uniq=-havazl>

# Annexe

Figure 3: Game Controller Component

Figure : Level Hierarchy

Figure 1: Unity Scene

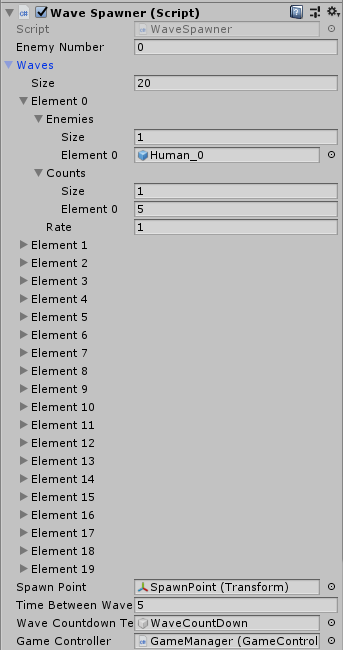


Figure : Wave Spawner Component

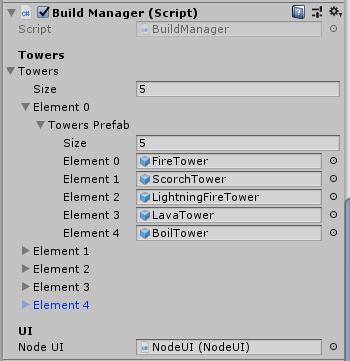
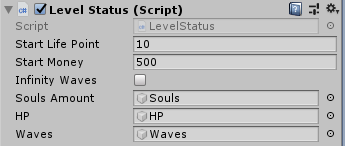


Figure 7: Nodes

Figure 6: Level Status Component

Figure 5: Build Manager Component

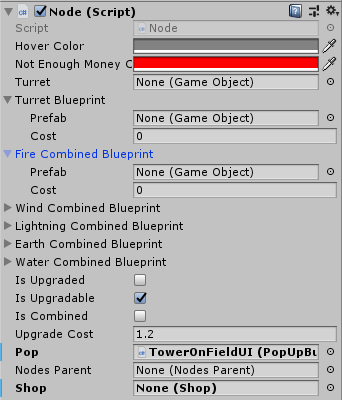


Figure 9: Shop UI

Figure 8: Node Component



Figure 11: Tower On Field Ui

Figure 10: Modify Tower Ui

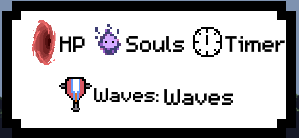


Figure 13: Indicator Panel

Figure 2: Menu Panel Ui

1. Une coroutine est un type de fonction qui n’attend pas un return avant de laisser une autre fonction s’exécuter, on peut donc appeler plusieurs coroutines qui s’exécutent selon le temps et selon un comportement donner [↑](#footnote-ref-1)
2. Régions qui contiennent tous les UIs. [↑](#footnote-ref-2)
3. C’est un principe de conception qui dit qu’une classe/fonction ne doit avoir qu’une seule responsabilité. [↑](#footnote-ref-3)
4. C’est un patron de conception créationnel qui permet d’instancier des objets dont le type est dérivé d’un type abstrait. [↑](#footnote-ref-4)
5. C’est un patron de conception comportemental qui a comme objectif de changer le comportement de l’état sans changer l’instance. [↑](#footnote-ref-5)
6. Le système de prefab permet de créer, configurer et de sauvegarder un GameObject compléter avec tous ses composants, les valeurs de ses propriétés et ses enfants (GameObject) comme un asset réutilisable. [↑](#footnote-ref-6)
7. Chaque GameObject peut avoir un tag, un peu comme une classe en HTML/css. [↑](#footnote-ref-7)
8. Un collider est une composante d’un objet qui permet de détecter quand un autre objet rentre en collision avec celui-ci. La forme et la taille d’un collider sont décidées par le programmeur à l’aide de l’interface de Unity. [↑](#footnote-ref-8)
9. Un particle system est une fonctionnalité de Unity qui permet la création d’effet visuel. [↑](#footnote-ref-9)
10. Une monnaie permettant d’acheter différents items (non implémenté). [↑](#footnote-ref-10)
11. Une classe qui sert de stockage [↑](#footnote-ref-11)