Rapport de projet - Jeu Unity - Lunatics

MIRADE Matéu BOURSIER Maxime PORTAIL Valentin
VAN DE MERGHEL Robin

2023

Table des matières

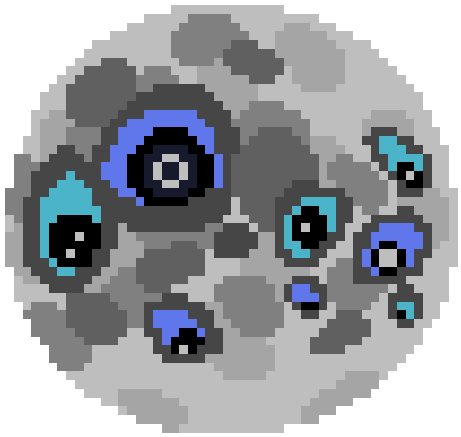

Figure 1: Lunatics - Icone du jeu

1 Présentation du projet

1.1 Introduction

Le projet "**Jeu Unity**", proposé par *M. Pastor* est un projet de programmation de jeu vidéo en groupe de 4 personnes. Il a pour but de nous faire découvrir le travail en groupe, et de nous faire découvrir le monde de la programmation de jeux vidéos.

Le but est d'avoir une première version d'un jeu vidéo, qui peut être améliorée par la suite. L'objectif étant d'apprendre à programmer un jeu vidéo, et non de faire un jeu vidéo complet, un jeu complet prenant des fois plusieurs années à être développé.

1.2 Présentation du jeu

Notre jeu est un **Tower-Defense** avec des phases de jeu d'exploration.

Nous pouvons définir ce qu'est un "Tower defense". C'est un type de jeu dont l'objectif est de défendre un objectif de vagues d'ennemis, traditionnellement à l'aide de tours, mais plus généralement à l'aide d'objets infligeant des dégâts aux ennemis. Ces objets peuvent s'acheter et généralement s'améliorer à l'aide d'une monnaie récoltée au cours des nuits.

La partie Tower-Defense se fait en *side-scrolling*. C'est-à-dire que la caméra est vue de côté et l'univers du jeu se voit ainsi en 2D. La plupart des jeux de plateformes sont des side-scroller.

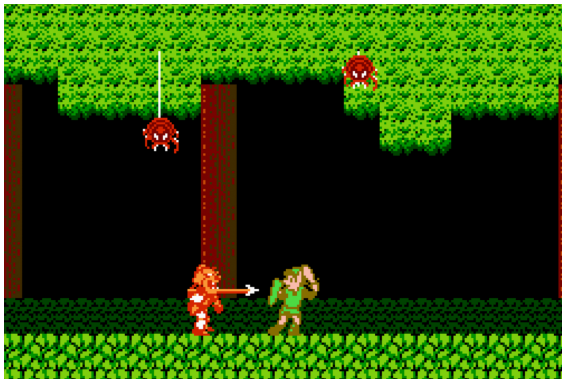

Figure 2: Zelda II: The Adventure of Link - Un jeu en side scrolling

1.3 Inspirations et principe

L'idée derrière le jeu pour ce qui est de la partie Tower Defense est largement inspirée par **Kingdom two Crowns**.

Figure 3: Kingdom Two Crowns

L'aspect et le gameplay sont largement similaires (*Tower Defense, Side-Scroller en 2D*). L'objectif de Kingdom Two Crowns est de conquérir des îles qui se font attaquer chaque nuit par des vagues de monstres. Pour se protéger et ensuite conquérir l'île, il faut étendre ses fortifications sur les deux côtés de l'île tout en se protégeant soi-même des vagues de monstres.

Kingdom Two Crowns a comme particularité de devoir *nécessiter des pièces* pour réaliser quasiment toutes les actions dans le jeu. Ces pièces se récoltent à l'aide de "fermiers" qui en génèrent en continu.

Lunatics fonctionne de manière assez similaire. Le but du jeu est chaque nuit de *défendre une centrale nucléaire futuriste*, et de *survivre à des vagues de Monstres*, appelés **Moonsters**, à l'aide de "Bâtiments" (nom donné dans le code : Building) qui vont les ralentir ou les tuer. Les bâtiments ont un *coût*, mais contrairement à Two Crowns, la monnaie ne se récupère pas à l'aide de générateur d'argent. Elle ne se récupère pas non plus en tuant les monstres, comme dans d'autres Tower Defense plus classique.

La monnaie se récupère dans **un monde de rêve**, auquel le personnage peut accéder à chaque lever du soleil en allant dormir.

Cette version du jeu n'est pour l'instant qu'une esquisse du jeu final. En effet, il manque un système de sauvegarde, et donc les bâtiments ne sont pas conservés entre les nuits.

1.4 Le monde de rêve

Le monde de rêve est l'une des parties cruciales du jeu. Là où durant la phase de nuit on doit défendre le centre de la carte en construisant des bâtiments, celle de rêve est la partie où l'on collecte de l'argent afin d'acheter des bâtiments. Une partie ne va pas sans l'autre.

Nous nous sommes globalement inspirés du concept de Yume Nikki, dont le principe est de se déplacer dans rêves d'une jeune fille afin de collecter des objets. Dans notre cas, les objets seront des pièces.

Figure 4: Yume Nikki

Cette phase s'articule sur de la recherche de pièces : le joueur doit, en se baladant dans le rêve, trouver des pièces réparties aléatoirement sur la carte, qui lui permettront la nuit suivante de pouvoir rajouter des défenses. Ceci diffère de la plupart des tower defense classiques, car la récupération des pièces n'est pas passive (récupération des pièces en tuant des ennemis ou générateur d'argent), le joueur doit explorer le monde généré afin de pouvoir progresser.

Le jeu est fait de telle sorte que, chaque nuit, la carte sera différente, et donc la répartition des pièces aussi. Comme la carte est générée aléatoirement avec différentes ambiances, le joueur reste stimulé tout le long de la partie : chaque ambiance est différente, et le joueur ne sait pas à quoi s'attendre. Ceci permet une rejouabilité accrue. Nous détaillerons le fonctionnement de la création de cette carte dans une prochaine partie.

2 Les graphismes

Les graphismes du jeu sont réalisés sur Piskel par *Matéu Mirade* et deux personnes extérieures au projet : *Willow Pages* et *Maëlle Duprat* (sur Photoshop au lieu de Piskel).

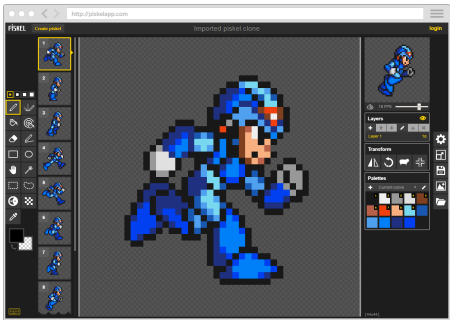

Figure 5: Piskel

Le dossier des sprites (Assets/Sprites) est organisé de la façon suivante :

- DreamSection
- Menus
- TowerDefenseSection

Avec les onglets DreamSection et TowerDefenseSection contenant les Sprites des sections de gameplay respectives. Menus quant à lui contient seulement les Sprites nécessaires à l'écran titre, de Game Over et de Pause.

2.1 DreamSection

DreamSection contient les objets collectionnables dans toutes les sections de rêve (`Assets/Sprites/DreamSection/Collectibles`) et chaque dossier correspond ensuite à un monde de rêve particulier. Bien qu'il y en ait qu'un seul dans cette version du jeu (`Assets/Sprites/DreamSection/CheeseCave`), toutes aurait le format suivant :

- Palette : Palette contenant tous les sprites nécessaire à la texturation de la map du monde de rêve
- Creatures : Dans une version plus avancée du jeu, aurait contenu les créatures vivantes du monde courant
- Background : Contient les textures du sol

Les sprites du monde de rêve n'ont pas de bordures noires, pour leur donner un aspect plus onirique par rapport aux éléments de la TowerDefenseSection. L'aspect d'un monde de rêve peut être très différent de l'un à l'autre.

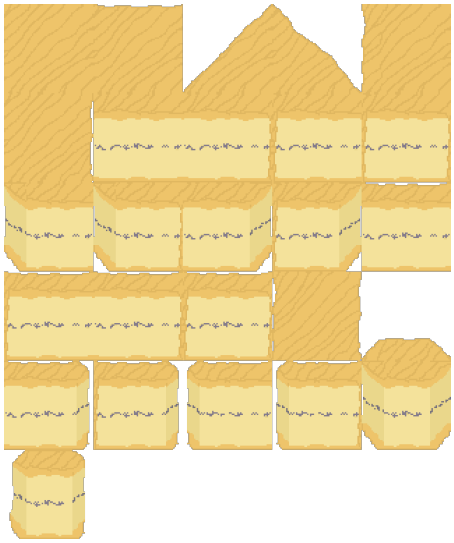

Figure 6: Palette du monde de rêve de la Cave à Fromage

2.2 TowerDefenseSection

TowerDefenseSection contient les éléments du monde réel du jeu. Ses dossiers sont :

- Background : Les fonds du monde, comme la Lune ou les montagnes par exemple.
- Buildings : Les Bâtiments (ceux construits par le joueur).
- Interactive : Les objets avec lesquels on peut interagir dans le monde étant ni un bâtiment, ni un Moonster, ni le joueur -`Moonster`: Les Moonsters (Moonlight et Eclipse)
- Player : Le Joueur et ses armes.
- PowerPlant : Les composantes de la centrale nucléaire.
- Terrain : La terre du sol
- UI : Barres de vie et menu des bâtiments.

Les sprites de la TowerDefenseSection ont tous des bordures noires, à l'exception des Moonsters et du Joueur afin de leur donner un aspect proche du monde de rêve.

Figure 7: Shepherd (avec des bords colorés) et Sentry (avec des bords noirs)

Les sprites se veulent cependant tous froids et assez moroses. Les tours et la centrale ont tous le même style graphique (gris avec des bordures jaune-cyan).

Nous n'avons par contre pas implémenté d'effets sonores et de musiques dans le jeu. Si l'on souhaîtait compléter le jeu, nous devrions en ajouter.

2.3 Menus

Les menus du jeu ont tous 3 options

- Menu de départ : Play, Reset et Exit
- Menu de Pause (accessible depuis la touche Echap) : Resume, Options, Exit
- Menu de Game Over : Retry, Reset, Exit

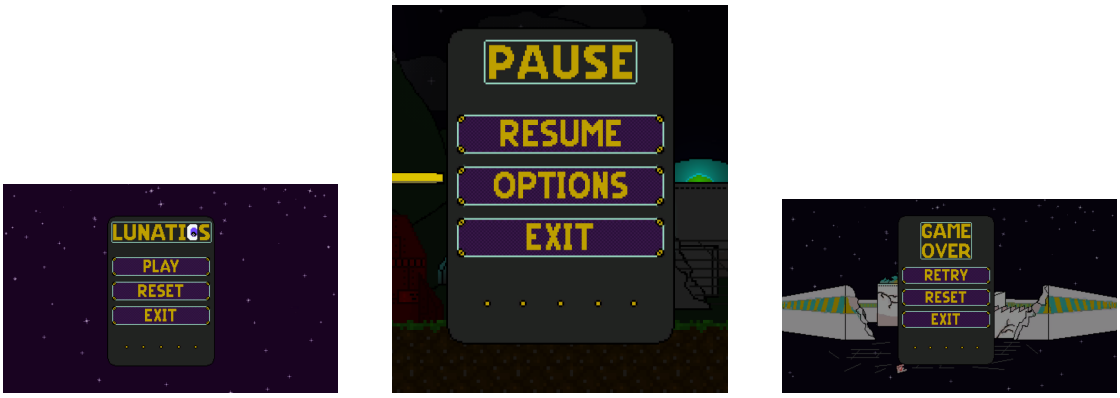

Figure 8: Écran Titre, Menu de Pause et Menu de Game Over

Les boutons Exit servent à retourner au menu de départ, sauf celui du menu de départ lui-même, qui permet de quitter le jeu.

Les boutons Play et Reset du menu de départ font la même chose. À un stade plus avancé du

développement, ils auraient permis de “reprendre la partie en cours” et “commencer une nouvelle partie”.

Le bouton Resume permet de quitter le menu pause, Options n’est pas implémenté mais aurait permis de changer ses commandes ou la résolution de l’écran par exemple.

Les boutons Retry et Reset du menu de Game Over font la même chose également. Ils auraient permis de reprendre depuis la nuit perdue et commencer une nouvelle partie.

3 Le code

3.1 Bâtiments

3.1.1 Rôle et caractéristiques des bâtiments.

Les bâtiments sont un point clé des towers defense, étant l’un des moyens principaux du joueur de se défendre face à des ennemis plus coriaces que lui-même. Ce sont des entités statiques servant à défendre la base.

Il existe actuellement deux bâtiments :

- les remparts : bâtiments purement défensifs servant à absorber un grand nombre d’attaques ennemies, mais ne possédant aucune faculté offensive. Il ne permet pas à lui seul de contrer une vague ennemie puisque étant incapable de se défendre, mais il permet de bloquer temporairement la progression des ennemis.
- les tourelles : bâtiments moins résistants qu’un rempart, mais dotées de facultés offensives. Elles permettent d’infliger des dégâts à des ennemis à portée. Ces tourelles sont les défenses les plus efficaces contre des ennemis, puisqu’elles infligent un grand nombre de dégâts, sans être facilement détruites.

Chaque bâtiment a un coût unitaire en pièces et occupe une certaine quantité d’espace une fois posé. Ces aspects servent à restreindre le joueur dans ses décisions vis-à-vis du placement des bâtiments.

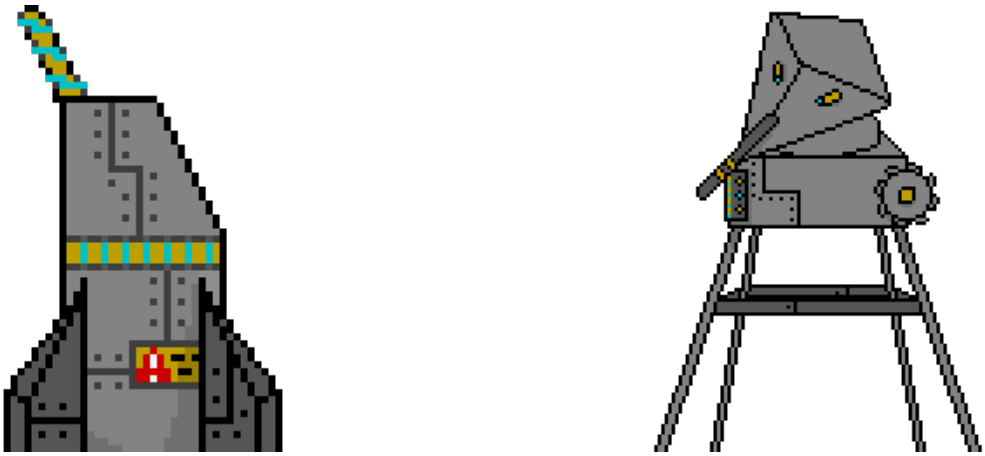

Figure 9: Rempart et Tourelle

3.1.2 Mécaniques des bâtiments.

Par défaut, aucun bâtiment n'est posé. Pour poser un bâtiment dans la zone de jeu, il faut que le joueur passe en mode construction, la touche attribuée étant `"B"`.

La gestion des bâtiments est faite par l'objet `BuildingManager`, auquel est attaché le script `BuildingManager.cs`.

Ce script permet d'une part de gérer les placements des bâtiments. Ce dernier utilise une `HashSet` contenant la position de chaque bâtiment actuellement posé, faisant de ces dernières des positions illégales pour poser d'autres bâtiments. Cette `HashSet` contient des entiers, représentant les coordonnées horizontales des bâtiments. Etant donnée le fait que tous les bâtiments sont terrestres et sont fixés sur un sol plat, la seule donnée révélant la position d'un bâtiment dans l'espace 2D est sa position sur l'axe des abscisses, d'où le choix d'une `HashSet` d'entiers.

D'autre part, ce script se charge de redessiner, à chaque frame, une zone de construction d'un potentiel bâtiment, si le mode de construction est actif. Cette zone permet de prévisualiser la position d'un futur bâtiment. Dans Unity, la position des objets est gérée par la composante `Transform`, qui représente notamment la position d'un objet en un vecteur à trois dimensions, avec le type `Vector3`. L'objet `PlacableTile`, composant `BuildingManager`, représente l'espace des bâtiments sur lequel on les place, avec une composante `Grid` qui, comme son nom l'indique, permet de représenter l'espace avec une grille de cellule.

`BuildingManager.cs` s'occupe de la conversion d'une position en `Grid` à une position en Vector3. Pour (x, y), les coordonnées entières d'une cellule dans une `Grid`, on convertit ce couple en un triplet $(x_M, y_M, 0)$, pour x_M et y_M, des entiers représentant les coordonnées en `Vector3` du centre de la cellule en question (on omet z_M ici car nous travaillons en deux dimensions).

La couleur de cette zone dépend de si elle survole une position valide (blanche) ou pas (rouge). Cette gestion des couleurs est faite par `PlacementIndicatorColorManger.cs`, toujours dans l'objet `BuildingManager`.

`BuildingManager` est aussi composé d'un second objet, étant `BuildingMenu`, gérant le menu d'achat des bâtiments. À ce dernier objet est attaché le script `BuildUI.cs`, permettant de gérer l'interface graphique du menu d'achat.

Chaque bâtiment est composé à la fois d'un script `Building.cs` généraliste, et d'un script particulier, par rapport au type du bâtiment. Chaque bâtiment a comme couche `"Building"` (`Layer`). Cette dernière est notamment utilisée dans le ciblage des bâtiments pour les ennemis.

Le script `Building.cs` contient tous les attributs d'un bâtiment : les points de vie (initiaux et actuels), les points de dégâts, le temps de latence entre deux attaques, ou bien le coût du bâtiment. `Buidling.cs` implémente `Damageable` qui définit la gestion des dégâts. Un attribut booléen de permet de rendre un bâtiment vulnérable ou non. Ce script gère aussi l'apparition d'un bâtiment par rapport à une position en `Vector3`, en plus d'effectuer le paiement pour le faire apparaître. La destruction d'un bâtiment est gérée par la fonction `Die` de `Damageable`.

Quant aux scripts particuliers, ils définissent le comportement d'un type de bâtiment auxquels ils sont associés.

Pour `Wall.cs`, il consiste juste à initialiser l'orientation du rempart par rapport à la base, le comportement d'un rempart consistant seulement à rester immobile.

`Sentry.cs` définit le comportement d'une tourelle par une boucle événementielle :

1. Recherche d'un ennemi à portée ;

2. En fonction de la présence d'un ennemi à portée, la tourelle attaque puis passe en attente d'une prochaine attaque, ou reste immobile.

`Sentry.cs` modifie des valeurs liées aux paramètres de l'`Animator`, un composant de Unity lié à la tourelle qui permet de gérer l'animation d'un sprite. Dans ce dernier, l'animation effectuée dépend de l'état de la tourelle dans l'`Animator`, similaire à un automate, et de paramètres, étant des attributs modifiables dans les scripts. Pour la tourelle, les paramètres sont les suivants : un booléen permettant de savoir si la tourelle attaque, et un flottant décrivant l'orientation du canon.

La tourelle, capable de cibler un ennemi, est composée d'un objet `Shooter`, contenant le script `ShootingZone.cs`, qui permet de gérer le comportement du ciblage des ennemis pour les bâtiments. Pour ce faire, ce dernier utilise une liste de `Collider2D`, un composant servant à représenter une zone de collision quelconque en 2D. Les zones de collisions détectées sont uniquement celles de la couche nommée `"Detect Moonster"`, qui n'inclut que des ennemis. Ce script permet de cibler, au choix, le dernier ennemi à être rentré à portée du bâtiment, plus performant puisque étant le premier élément de la liste, ou l'ennemi le plus proche du bâtiment, étant à portée. La portée est représentée par un `CircleCollider2D`, définissant une zone de collision sphérique en 2D.

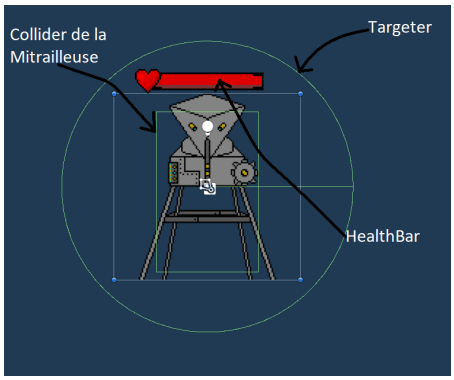

Figure 10: Schéma de la tourelle

3.2 General Components

Les Moonsters, les bâtiments et les armes (objets peu développés dans notre programme) ont toutes prévues d'avoir des caractéristiques en commun. De manière traditionnelle dans un langage objet, une solution est l'héritage. Sauf que l'héritage dans notre situation, c'est-à-dire en travaillant avec des `MonoBehaviors` sur Unity, est une solution très difficile à mettre en place pour un gain minime. En effet, et c'est ce qui a été fait dans notre jeu, on peut simplement attacher un script `Moonster`, `Building` ou `AbstractWeapon` à notre objet et puisque les objets interagissent entre eux par `GetComponent<>()`, il nous a semblé plus simple d'attacher ces scripts plus généraux et de les référencer dans d'autres objets qui ont de toute façon que rarement besoin du composant particulier.

3.3 Les Moonsters

3.3.1 Caractéristiques des Moonsters

Un jeu de Tower Defense se doit d'avoir son lot d'ennemis. Ce jeu n'échappe pas à la règle. Les Moonsters, contraction entre Moon et Monster, ont pour objectif de prendre le contrôle de la centrale électrique que vous essayez de défendre.

Dans le code, les Moonsters possèdent certaines caractéristiques :

- Une puissance (`Power`), un nombre arbitraire qui sera utilisé ultérieurement pour générer les vagues.
- Une vitesse (`Speed`), qui désigne la vitesse de déplacement du Moonster.
- Une portée d'attaque (`AttackRange`), qui indique la distance sur laquelle un`Moonster`peut attaquer un autre objet.
- Une puissance d'attaque (`AttackPower`), qui précise le nombre de points de vie pouvant être retirés par le`Moonster`à chaque attaque.
- Un temps de récupération entre les attaques (`AttackCooldown`).
- Les points de vie (`InitialHealth`) du Moonster

Il existe deux espèces principales de Moonsters :

- Le Dunkel (sombre en allemand) est un petit ennemi faisant peu de dégâts, mais qui peut devenir redoutable s'il est accompagné de plusieurs autres Dunkels. Les bâtiments, qui bloquent sa trajectoire, est sa principale faiblesse.
- Le Mhund (contraction entre Moon et Hund, chien en allemand) est un ennemi quadrupède, plus grand que le Dunkel, plus rapide, plus puissant et surtout capable de sauter. Il est donc capable de passer par dessus la défense du joueur pour directement attaquer la centrale.

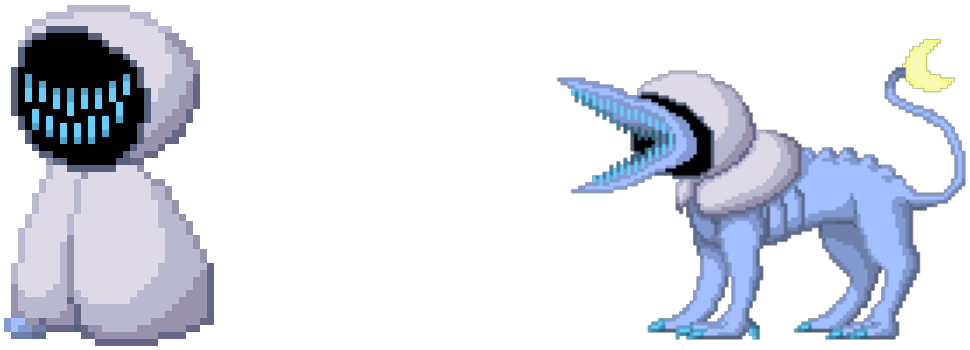

Figure 11: Dunkel et Mhund

Chacun de ces Moonsters est décliné dans une version Eclipse, avec des capacités améliorées, ce qui permet de pimenter un peu les parties.

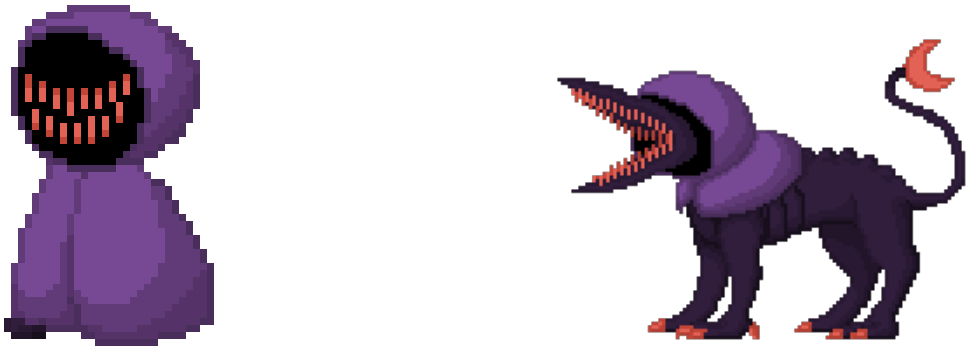

Figure 12: Dunkel Eclipse et Mhund Eclipse

Voici un tableau résumant les caractéristiques de chaque Moonster, versions Eclipse incluses. En gras sont indiquées les améliorations des versions Eclipse :

Nom	Dunkel	Eclipse Dunkel	Mhund	Eclipse Mhund
Power	1	3	10	10
Speed	3	3	4	4
AttackRange	0.5	0.5	1	1
AttackPower	6	6	20	50
AttackCooldown (secondes)	0.8	0.5	1	1
InitialHealth	100	250	500	1000

3.3.2 Comportement des Moonsters

Dans un jeu de type Tower Defense, les ennemis doivent avoir un comportement qu'il est possible de prévoir, afin de pouvoir placer correctement les bâtiments. En général, ils se contentent d'avancer "tout droit" pour atteindre leur objectif, tout en infligeant des dégâts aux bâtiments qu'ils croisent sur leur route.

Ici, le comportement des Moonsters respecte globalement cette logique. En effet, leurs déplacements se font en général sur un axe horizontal. Ils commencent d'abord par repérer l'objet le plus proche d'eux dans un certain rayon, que ce soit un bâtiment, le joueur ou bien la centrale électrique. Chaque`Moonster`a un rayon de portée différente. S'ils ne trouvent pas de cible dans leur rayon de portée, ils avancent vers la centrale.

Une fois la cible repérée, le`Moonster`avance vers elle et lui inflige des dégâts dès que possible.

D'un point de vue technique, cela fonctionne avec les objets `Vector2` intégrés à Unity. À chaque instant, le jeu va calculer la distance entre le`Moonster`et trois objets présents sur le terrain : la centrale, le joueur et le dernier bâtiment posé.

3.3.2.1 Cas du Mhund Le Mhund par rapport aux ennemis, a un comportement un peu différent. En effet, il est capable de sauter par dessus les bâtiments pour atteindre la centrale :

il ignore les bâtiments contrairement au Dunkel par exemple. La seule situation où il se met à attaquer un bâtiment est celle où le joueur se cache en dessous d'un bâtiment.

Pour pouvoir détecter si un joueur se cache en dessous d'un bâtiment, on utilise la méthode `Raycast` du module `Physics2D`. On peut voir cela comme un rayon envoyé par le`Moonster`vers son éventuelle cible et qui détecte le premier obstacle traversé. Cela permet de trouver le premier objet dans la zone d'attaque du Moonster. S'il est trouvé, l'objet devient la nouvelle cible du Moonster. Sinon, il vise la centrale.

3.3.3 Mécaniques du `WaveSpawner` :

Maintenant que nos Moonsters ont un comportement, il faut les faire apparaître sur la scène. Pour cela, deux scripts sont présents :

- `MoonsterManager` va contenir des références vers les différents types de Moonsters.
- `WaveSpawner` va générer à partir de ces Moonsters une vague complète et va l'invoquer petit à petit sur le terrain.

Cependant, nos vagues ne peuvent pas être générées n'importe comment. Il faut qu'elles répondent à certaines contraintes. Elles doivent notamment être de difficulté croissante, ce qui fait que le nombre de Moonsters devant apparaître va varier en fonction du numéro de la nuit.

De plus, certains types de Moonsters, notamment les versions Eclipse, ne doivent apparaître qu'au bout d'un certain nombre de nuits. Enfin, les vagues ne doivent pas être stockées "en dur" dans le jeu et les critères doivent pouvoir facilement être modifiés à l'avenir afin de rééquilibrer le jeu s'il est trop facile ou trop difficile.

La création des vagues se décompose en deux étapes : la composition de la vague, puis l'invocation progressive des Moonsters qui la composent.

La vague sera générée de manière aléatoire afin de varier les parties. Cependant, afin de garder une certaine consistance, une puissance totale sera attribuée aux vagues. Pour l'instant, elle est égale au numéro de la nuit multiplié par 10. Dans l'idéal, elle devrait être la somme des puissances des Moonsters qui la composent.

De plus, chaque`Moonster`possède sa propre puissance :

Nom	Dunkel	Eclipse Dunkel	Mhund	Eclipse Mhund
Power	1	3	10	10

Tant qu'il reste des Moonsters à placer dans la vague, on en génère un aléatoirement. Si sa puissance ne fait pas dépasser celle de la vague, on l'ajoute. Sinon, on en génère un nouveau.

Jusqu'à la nuit 12, on ne pioche que parmi les Moonsters classiques. Dès la nuit 13, on peut ajouter les Moonsters Eclipse. Tous ces paramètres (puissance d'une vague, apparition des versions Eclipse) peuvent facilement être modifiés.

Ensuite, chaque`Moonster`de la vague apparaît un à un dans un des coins du terrain (à gauche ou à droite) avec un intervalle entre chaque`Moonster`de 1 à 10 secondes.

De cette manière, il est possible de créer des vagues consistantes entre chaque partie tout en créant un sentiment de surprise chez le joueur car chaque partie sera différente.

3.4 Barres de vies

Les classes Building, Moonster, Player et Objective ont toutes un point commun : elles peuvent être blessées. C'est pour cela que l'on a une interface `Damageable` qui permet d'unifier tous les différents aspects du code de gestion de point de vie et de généraliser (un`Moonster`peut vouloir attaquer Objective, Player ou Building. Au lieu de réécrire 3 fois le même code, on peut lui demander de récupérer la composante `Damageable` de sa cible).

Les barres de vies sont toutes régies par un même script `HealthBar`. Ce script fonctionne en combinaison avec la classe Damageable. Celle-ci a comme propriété `miniSpriteRenderer`, qui permet à `HealthBar` de savoir quel sprite utiliser pour son icone. `HealthBar` a également un attribut color pour la couleur de la barre.

La barre de vie en elle même est juste un `[Slider](https://docs.unity3d.com/2018.2/Documentation/ScriptRefe` qui règle en fonction des PVs du `Damageable` la largeur du rectangle.

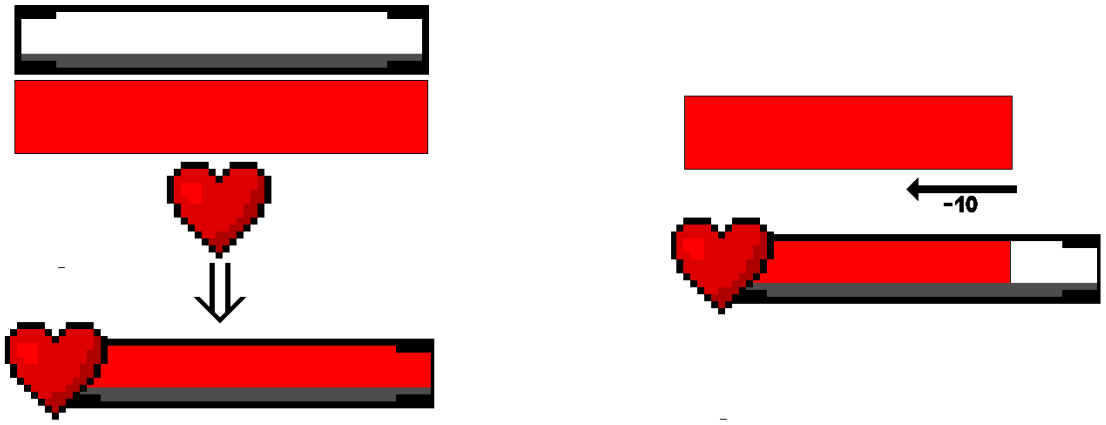

Figure 13: Barre de vie d'un Moonster et d'un Building

3.5 Génération procédurale de la carte

Comme dit précédemment dans la présentation du jeu, durant la phase de jour, le joueur va évoluer dans une carte en deux dimensions générées aléatoirement à chaque fois. La façon de générer cette carte, est appelée "Génération procédurale".

La génération procédurale est une méthode de création de données (ici une carte) de façon algorithmique : on se pose des règles et des contraintes, et on doit trouver la suite d'instructions optimale pour se rapprocher de l'objectif.

Dans notre cas, on avait les contraintes suivantes :

- Une carte en deux dimensions
- Contient des grottes de tailles variables
- Il n'y a en fait qu'un seul réseau de cavités, car reliées par des tunnels
- Un tunnel ne doit pas en croiser un autre, et ne doit pas couper une grotte
- Garder une esthétique dans la forme globale
- Garder en mémoire les cases disponibles du réseau afin de pouvoir y placer des pièces à posteriori

De plus, comme chaque partie de notre code, on a décidé de rendre modulable le code : taille de la carte, taille minimale et maximale des cavernes programmables ainsi que l'épaisseur des tunnels. Tout cela se programme dans le Manager prévu à cet effet dans l'éditeur de scène de Unity.

Par rapport à la partie algorithmique même, on suit cinq étapes :

- Générations de trous aléatoires
- Pré-processing avec filtrage
- Identification des différentes cavités (aka Rooms)
- Fusions des cavités (aka Rooms)
- Post-processing avec filtrage

Nous avons décidé de partir d'une carte pleine. Cela permet de "creuser" sans la carte, plutôt que d'essayer de générer les contours d'une grottes. C'est un gain en facilité.

Nous "creusons" dans la carte à sa génération : un générateur aléatoire, qui prend en paramètre un fillRate (pourcentage de remplissage de la carte) remplit une certaine partie de la carte de cellules. À la fin, on a une carte trouée.

Suite à cela, on va affiner la carte avec une fonction de Smoothing : si une cellule a moins de quatres voisins, on la supprime, sinon elle survit. Cela creuse vraiment dans la carte, et élimine les trous isolés. On applique cette fonction un nombre déterminé de fois que l'on configure. On obtient alors une carte percée.

Ensuite, un code va, à partir de la grille obtenue, séparer les différentes cavités (aka Rooms). Pour cela, il y a une recherche récursive de voisins. On obtient alors une liste de cavités, c'est-à-dire une liste de liste de cellules vides. On notera que l'on stocke l'ensemble des cellules vides et la liste des bords de la cavité.

Cette dernière liste qui contient les bords de la cavité permet de pouvoir déterminer la distance entre deux cavités : on recherche les cellules les plus proches entre deux cavités, puis on calcule la distance entre ces deux cellules.

Maintenant qu'on a toutes les informations importantes, on applique l'algorithme suivant :

```
Tant que le nombre de cavité est différent de 1 :
    On récupère la première cavité que l'on appelle C1
    On trouve la cavité la plus proche de C1, on la note C2
    On fusionne C1 et C2, on stock le résultat dans C3
        On n'oublie pas de creuser un tunnel entre C1 et C2
    On remplace C1 par C3
    On supprime C2
```

Après n itérations, n représentant le nombre de Rooms, on obtient alors une seule et même Room, qui ne se croise jamais (comme on relie toujours les Rooms les plus proches). Cette Room peut garder une esthétique en changeant les paramètres, et contientla liste des cellules disponibles.

On a bien n itérations, car à chacune, on décrémente le nombre de Room de 1.

Au début, on avait que deux types de textures à appliquer : vide si la Room est vide, sinon une texture temporaire. Ensuite, avec un traitement post-processing, on applique du texturing plus intelligent.

On ajoute des textures plus complexes, qui ajoutent alors du relief dans la scène. Cela projette le joueur dans de la 2.5D au lieu d'une 2D classique, à l'instar de pokémon sur les anciennes

versions : on n'est pas totalement de profil, il y a une légère perspective.

Pour cela, on applique l'algorithme suivant :

```
Pour chaque case pleine
    On regarde ses voisins :
        On initialise une valeur qui représente comment est entourée la case : bx,y
        En fonction des voisins, on ajoute des poids qui représentent les voisins
        En fonction de la valeur finale bx,y, on importe un sprite différent
            (cellule pleine, coin droit, …)
```

À la fin on obtient l'affichage graphique de la carte suivant :

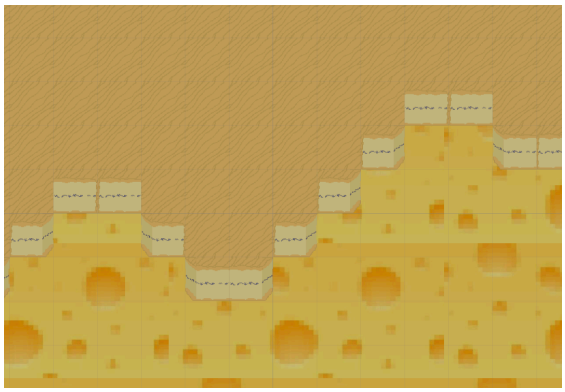

Figure 14: Rendu de la Dream Section après traitement

4 Conclusion

Le but du projet était de découvrir différentes façon de programmer : entre language (ici C# avec Unity) et travail de groupe.

En effet, nous avons pu découvrir le travail de groupe, et les difficultés que cela implique. En effet, il faut savoir communiquer, et se mettre d'accord sur les choix à faire. Le plus dur était de prendre des décisions. Cela peut être difficile, mais c'est une expérience enrichissante.

4.1 Améliorations possibles

Comme le projet n'était qu'une esquisse, il y a beaucoup d'améliorations possibles. Nous avons déjà parlé de la sauvegarde des bâtiments, mais il y a d'autres choses que l'on peut améliorer :

- Ajouter des bâtiments
- Ajouter des Moonsters
- Ajouter des niveaux de difficulté
- Ajouter des effets sonores et de la musique
- Créer plus de monde de rêve
- Révision et ajouts d'armes
- …

4.2 Remerciements

On remerciera fortement nos designeurs, qui ont fait un travail remarquable sur les sprites en aidant *Matéu Mirade* : *Willow Pages* et *Maëlle Duprat*. Sans eux, le jeu n'aurait pas été aussi beau.

On remerciera aussi *M. Pastor* pour nous avoir guidé tout au long du projet, et nous avoir orienté dans nos choix.