Rapport de soutenance

Bloqueurs convexes

Thomas SIRVENT
William GROLLEAU
Adrien ANTON-LUDWIG
Eliès BILLOTTA

Table des matières

1 Introduction

Dans un premier temps, ce rapport présentera globalement le projet, les idées que nous avons eues ainsi que le jeu que nous souhaitions créer il y a quatre mois. Ensuite, nous présenterons les différents membres du groupe.

Afin de simplifier un maximum la compréhension de ce qui a été fait, nous avons décidé de séparer les tâches produites en deux parties.

Tout d'abord, la réalisation du projet dans laquelle nous présenterons la totalité du travail qui a été fait pour chacune des parties du projet, il s'agira d'expliciter ce qui a été réalisé et qui est présent dans le jeu.
Dans un second temps, viendra la conception du projet, qui s'attardera sur la manière dont chacune des parties a été faite, les difficultés rencontrées ainsi que les solutions déployées pour y parer.

Nous exprimerons ensuite l'expérience de chacun, le ressenti sur ce projet et ce que l'on a réussi à en tirer. Elle sera divisée en quatre pour que chacun des membres puisse faire le bilan de ce qu'il a appris lors de ce projet.

Nous finirons par une conclusion plus générale sur l'entièreté de ce projet, que ce soit sa réalisation, sa conception ou encore le ressenti général du groupe sur cette épreuve que nous avons passée.

2 Présentation du groupe

2.1 Les Bloqueurs Convexes

Le groupe s'est créé assez naturellement étant donné que nous étions déjà amis. Après quelques discussions, nous sommes rapidement tombés d'accords sur le thème d'un jeu que nous aimerions développer. Par la suite, nous avons fait le point sur nos compétences et remarqué qu'elles étaient suffisamment diversifiées pour mener à bien ce projet. Nous en avons donc conclu que cette collaboration ne pouvait être que bénéfique. Concernant le nom du groupe, nous n'avions aucune idée, mais ce n'était pas le cas du générateur de mot aléatoire, qui nous a proposé "Bloqueurs Convexes" !

2.2 Membres convexes

Thomas SIRVENT (Chef de projet)

J'aime bien les jeux vidéos comme pas mal de monde à EPITA, et c'est donc avec joie que j'ai saisis l'opportunité d'en faire un. Je me suis déjà amusé à faire un ou deux prototypes de jeu dans le passé, mais cette fois nous sommes plusieurs sur le projet, et c'est du sérieux. J'espère gagner de l'expérience en matière de gestion/travail de groupe au cours de ce projet, ainsi que de passer un bon moment ! Nous avons beaucoup d'idées et toutes les mettre en place sera un défi, mais je pense que nous pouvons le surmonter !

> *"Il suffit d'ingurgiter une poêlée de petites pommes de terre sautées pour venir à bout de tous les défis"* – Platon

William GROLLEAU

Avant Epita, mes premières expériences en programmation se résument à un Flappy ~~Bird~~ Turtle sur téléphone. Ce projet est pour moi l'occasion de grandement m'améliorer en programmation ainsi que d'acquérir une expérience concrète de ce que représente un projet, que ce soit le travail d'équipe, les timelines à respecter, les rush, le stress et tout ce qui accompagne un projet digne de ce nom. Nous sommes tous très impliqués dans le projet, nous avons beaucoup d'idées et elles convergent toutes. Ce projet sera certainement de taille, mais je suis assuré que nous parviendrons à avoir un rendu qui nous convient.

> *"I am not a goddamn Turtle"*– TurtleSmoke

Adrien ANTON LUDWIG

J'ai depuis de nombreuses années apprécié de nombreux jeux vidéos... et j'y ai consacré bien plus que de nombreuses dizaines d'heures... Le temps est enfin venu de changer de rôle, passer de joueur à développeur ! D'une part, ce projet va venir renforcer mon expérience de travail en équipe. Expérience qui me sera utile tout au long de ma vie en tant qu'ingénieur. D'autre part, je serai amené à effectuer un grand nombre de recherches et à emmagasiner beaucoup de connaissances pour mener à bien le projet. Ce sera donc une expérience très enrichissante, que j'ai hâte de réaliser.

> *"Real programmers don't comment their code. It was hard to write, it should be hard to understand."*– An anonymous saint

Eliès BILLOTTA

Les jeux vidéos ont toujours tenu une place importante dans ma vie. Étant un joueur régulier, j'ai toujours voulu connaître le fonctionnement de mes jeux favoris, apprendre à les reproduire et même créer mes propres jeux vidéos. Grâce à ce projet, j'ai aujourd'hui l'opportunité de réaliser ce souhait avec mes camarades de classe. Efficacité, collaboration et réussite seront à mon sens les 3 valeurs que nous veillerons à respecter tout au long du projet. Et qui plus est, tout au long de notre carrière d'ingénieur. Ma motivation n'ayant d'égal que ma curiosité dans ce domaine, je suis donc impatient à l'idée de voir notre jeu terminé et complet.

> *"Do or not do. There is no try."*– Master Yoda

3 Présentation de Tesseract

3.1 Origine et concept

Initialement nous avions plusieurs jeux et concepts en tête. Après concertation, nous nous sommes mis d'accords sur l'idée de faire un **Rogue-like**, c'est-à-dire une exploration de donjon dans un univers bi-dimensionnel généré aléatoirement. Ce choix semblait judicieux car chacun avait une partie intéressante qui lui plaisait et personne n'était forcé de faire quelque chose qui lui semblait désagréable à effectuer. C'est ainsi que nous avons eu l'idée de développer ensemble le jeu **Tesseract**.

Trois jeux du même genre ont été une source d'inspiration pour notre projet : **Black Futur 88**, **Hyper Light Drifter** et **Hammerwatch**. Tous ont des gameplays intéressants ainsi que des textures très innovantes qui nous ont beaucoup inspiré lors de la création de notre projet.

3.2 Histoire

Vous vous réveillez dans une salle, plongée dans l'obscurité. Depuis combien de temps êtes-vous prisonnier de ces murs ? Des semaines, des mois, des années peut-être ? Les couloirs autour de vous semblent sans fin, plongeant dans l'obscurité. Au milieu des cris stridents des créatures hantant ces lieux, une voix douce et intrigante se démarque, celle de Xelia, votre guide. Sur ses conseils commence alors une aventure terrifiante et tumultueuse. Prenez les armes. Usez de mémoire pour ne point vous perdre dans ce labyrinthe mouvant et triompher des monstres. Saurez-vous percer les énigmes entourant ce monde ainsi que le mystérieux **Tesseract** ?

3.3 But et intérêt

L'esprit de notre jeu est celui d'un jeu dynamique avec un rythme rapide. Avec peu de temps morts, le joueur n'aura que de rares pauses. Il devra faire vite et n'aura pas de seconde chance.

Malgré le côté frénétique du gameplay, notre jeu sera aussi tactique. Le joueur devra réfléchir, user de sa mémoire et de stratégie s'il ne veut pas se faire dévorer par les monstres de la carte, en perpétuel changement.

Guidé par la voix de la mystérieuse Xelia lors de son aventure, le joueur découvrira plus en détail l'histoire du monde dans lequel il se trouve.

Enfin, notre souhait serait que le jeu ait une dimension compétitive mais que les joueurs passent un bon moment avant tout !

4 Réalisation

4.1 Réseau

Dans Tesseract vous pouvez jouer jusqu'à quatre de vos amis. Les utilisateurs peuvent créer des salles de jeu, les rejoindre et lancer une partie. Une partie consiste en un donjon d'un seul étage, avec un boss à la fin. A la différence du mode solo, le temps nécessaire pour finir le donjon est chronométré. Lors de la complétion de ce donjon, le temps ainsi que la graine utilisée pour la génération de la carte sont envoyés au serveur afin d'établir un classement. Ainsi, les joueurs ont le choix de tenter de battre un record existant ou générer un nouveau niveau aléatoire. Le serveur de jeu réalisé en C# s'occupe de gérer l'authentification des joueurs, la création des parties ainsi que des parties en elle-même.

4.2 Site web

Le site web réalisé sous le framework Vue.js possède une courte présentation du jeu, une liste des membres et des ressources utilisées, ainsi qu'un lien pour télécharger l'installateur du projet (client, serveur, rapport de soutenance)... Il est possible de créer un compte pour pouvoir ensuite jouer dans le jeu en multijoueur. Une page de profil est ainsi créée, sur laquelle les différentes informations du joueur apparaissent (rang, en ligne ou non actuellement, récentes parties..). Le site possède aussi une section "wiki", dans laquelle les informations sur les différents éléments du jeu sont répertoriées.

4.3 Textures

Les textures de notre jeu se veulent dans un style "pixel-art" et rendent hommage à la plupart des jeux rétros et old school des débuts des jeux vidéos, comme Donkey Kong ou encore Pac-Man. Ce style s'impose comme une solution de luxe afin d'obtenir des personnages et des décors d'une qualité originale. Étant novices dans le graphisme, cette solution nous semblait des plus convenable pour arriver à un résultat à la hauteur de nos attentes.

Ainsi, murs, personnages et ennemis sont tous issus de notre imaginaire collectif et sont uniques à notre jeu. Chaque personnage reste cependant inspiré de licences fortes qui nous ont tous plus ou moins marqué dans notre culture jeuvidéoluqiue, de The Legend of Zelda en passant par The Binding of Isaac à Final Fantasy. Tous les personnages restent facilement identifiables, respectant des codes basiques ; le mage dégage un côté mystérieux, l'assassin

possède un shuriken... Le style n'est sûrement pas novateur mais se veut efficace et simple. Il en est de même pour les animations, qui restent uniques et simples.

Enfin, durant le projet, nous avons réussi à recréer une ambiance de donjon convenable et classique, tout en l'adaptant à notre style. En plus des jeux de lumières qui donnent une atmosphère ténébreuse à notre jeu, nous avons essayé de reproduire au mieux des éléments significatifs présents dans des châteaux ou donjons ; carrelages, murs abîmés, pièges, coffres, tables et autres éléments contribuent à une identification simple pour le joueur, qui se situe dès son arrivée en jeu dans un lieu obscur où il devra braver tous les dangers.

4.4 Mécanique de base

Les mécaniques de base sont les premiers composants nécessaires pour avoir un jeu jouable, en effet sans cela, nous ne pourrions pas déplacer notre personnage, attaquer ou encore utiliser des potions, en réalité nous n'aurions même pas de personnage.

Cette partie s'occupe des quatre personnages jouables, c'est-à-dire qu'elle s'occupe de gérer leurs déplacements, leurs attaques, leurs textures, leurs animations et plus généralement tout ce qui concerne le joueur. Elle est également en charge du système d'expérience permettant au joueur d'évoluer et donc de gagner en caractéristiques, mais également de débloquer des points de compétences lui permettant d'améliorer celles-ci.

Les mécaniques de base concernent également l'interaction avec l'environnement, elle permet au joueur d'interagir avec les coffres, d'utiliser des portails de téléportation, ainsi que d'activer les pièges se trouvant au sol lorsque le joueur est suffisamment proche. Elle va également permettre au joueur d'utiliser des potions afin de profiter des effets bénéfiques de celles-ci.

Elle s'occupe également de gérer les changements de niveau, d'un étage à l'autre, de la sélection du personnage ainsi que des sauvegardes automatiques disponibles après avoir vaincu un boss. Elle s'occupe également de permettre au joueur de changer ses touches s'il le souhaite.

4.5 Interface

L'interface du jeu se veut des plus intuitive. L'ATH, c'est-à-dire l'ensemble des informations visuelles visibles à l'écran, d'un design sobre et efficace, a été pensé pour ne pas surcharger en informations le joueur. Des éléments simples sont donc utilisés, avec le strict nécessaire : barres de vie, mana, expérience et un inventaire. Toutes ces informations s'actualisent en fonction de l'état du joueur. Par exemple, si le personnage subit des dégâts d'un ennemi, sa santé diminuera et cette information sera visible sur l'écran via la barre de vie qui diminuera en taille. Il en est de même pour les autres barres de ce même type. Pour ce qui est de l'inventaire, ce dernier se met à jour à chaque consommation ou récupération de potions par le joueur.

Concernant les différents menus du jeu, ils suivent cette logique intuitive entreprise par l'ATH. Le premier menu du jeu est le menu d'accueil. Établissant le premier contact entre le jeu et le joueur, il permet de sélectionner le mode offline ou online du jeu. En cliquant sur le mode "offline", le joueur a directement accès au jeu et se déplacer dans le donjon. Concernant le mode online, ce dernier est accessible depuis le meu d'accueil via le bouton du même nom. Le joueur accède alors à un second menu, dans lequel il peut se connecter à des rooms de jeux ou encore discuter avec ses partenaires grâce à un tchat.

En plus du menu d'accueil, trois autres écrans sont disponibles en dehors des scènes de jeu. En cliquant sur le bouton pause, accessible directement depuis la scène du jeu, le joueur peut accéder à un menu intermédiaire et mettre en pause le jeu ou, éventuellement, le quitter.

Enfin, le menu Game Over apparaît lorsque le joueur perd la partie. Il n'a alors le choix que de quitter le jeu et retourner au menu d'accueil, où il pourra à nouveau relancer une partie.

4.6 Objets

Pour enrichir l'aventure et rajouter des mécaniques intéressantes, qui complètent les capacités du personnage principal, nous avons créé une sélection d'objets. Elle est composée d'armes, associées aux différentes classes du personnage ainsi que d'objets déclenchables, des potions, que le héros peut transporter en quantité limitée.

Les potions sont des objets éphémères. Cela signifie que le joueur perdra celles-ci à chaque mort. Et qu'il devra ainsi se battre pour en récupérer à chaque nouvelle vie. Elles sont aussi consommables, c'est-à-dire qu'elles sont à usage unique. Une fois la potion activée, ses effets sont appliqués et cette dernière est détruite. Aussi, il existe trois niveaux de rareté par potion. Une potion plus rare a bien entendu des effets plus puissants.

Les armes au contraire seront permanentes, c'est-à-dire qu'elles resteront dans l'inventaire du joueur après sa mort. En effet, la puissance surnaturelle de celles-ci les lie au joueur. Ils lui permettront de s'améliorer sur le long terme, contrairement aux potions à usage unique et effet limité dans le temps. Ainsi, le joueur pourra améliorer son arme tout au long de la partie.

Les objets peuvent être récupérés de deux manières différentes. D'une part, les potions sont contenues dans des coffres, générés aléatoirement sur la carte, qui sont eux aussi déclenchables et à usage unique. Ainsi, la quantité de potions est limitée, il faudra donc que le joueur les utilise à bon escient. D'autre part, les armes sont récupérables sur le corps des ennemis vaincus. Le joueur devra donc être vaillant et affronter les ennemis pour accéder à celles-ci.

4.7 Carte

La carte agit sur deux plans, d'abord un plan théorique dans lequel elle va placer tous les éléments aux bonnes positions, elles placent donc les murs, les sols, les coffres et plus généralement toutes les décorations, elle doit également s'assurer que la carte est cohérente, que les décorations ne bloquent pas l'entrée d'une salle et que toutes les salles soient bien reliées en elles. Il faut également vérifier que l'accès à la salle du boss soit possible.

Elle s'occupe ensuite s'occupe d'appliquer les bonnes textures aux bons éléments et elle s'assure que tout reste cohérent, en effet, certaines textures ont besoin d'être positionnées en fonction d'autre élément pour avoir le rendu

visuel voulu. Une fois les textures assigner, elle s'occupe également d'ajouter un effet d'ombre et de lumière sur le terrain pour donner un aspect plus réaliste et plus beau au jeu.

Enfin, la carte va ajouter dans les coffres les objets qu'ils doivent contenir, envoyer à l'intelligence artificielle les emplacements libres pour lui permettre de générer les monstres, mais également envoyer les informations nécessaires pour placer le joueur sur le terrain.

4.8 IA

L'intelligence artificielle (IA) de notre jeu consiste à gérer les ennemis. Tout d'abord, elle s'occupe de les générer en fonction de l'environnement et du niveau du joueur. Pour proposer au joueur une expérience constamment intéressante, il est indispensable d'adapter le jeu à ses performances pour le pousser à toujours progresser. Ainsi, le niveau des ennemis s'adapte au niveau du joueur tout au long de la partie. Chaque fois que le joueur réalise de belles performances et gagne un niveau, les ennemis deviennent plus puissants, plus rapides et plus résistants.

Ensuite, elle prend en charge les déplacements des ennemis à travers la carte, jusqu'au joueur. En effet, elle calcule le chemin le plus court entre ceux-ci en fonction des contraintes de l'environnement de la carte. Évidemment, l'ennemi ne rentre en collision avec aucun obstacle sur son trajet. Il s'agit là du principe de "pathfinding" classique que nous avons adapté afin que les ennemis se comportent de manière cohérente en présence de plusieurs joueurs.

Enfin, elle contrôle aussi les attaques et veille à ce qu'elles soient utilisées à bon escient, encore une fois en tenant compte de l'environnement. Cela signifie que les attaques au corps à corps ne sont utilisées qu'une fois à portée du joueur. Aussi, les attaques à distances ne sont pas utilisées si un obstacle est présent entre l'ennemi et le joueur. Autrement dit, l'IA s'assure que les ennemis n'utilisent pas leurs capacités dans des situations de non sens.

5 Conception

5.1 Réseau

Le serveur est responsable de recevoir les demandes d'authentification des clients. En effet, lors de la connexion en ligne il faudra entrer le mot de passe de son compte. Le serveur permet aussi d'échanger avec la base de données, afin d'accéder aux différentes cartes disponibles en ligne et leurs classements. Il gère également les salons de jeu, que ce soit le listage de ceux-ci ou de leurs paramètres, comme le nombre de joueurs ou la carte de jeu. Puis, il envoie la graine de génération de la carte aux clients des joueurs et instancie une nouvelle partie. Cette instance, vers laquelle seront dirigées les requêtes des différents clients, reçoit et redistribue les actions des joueurs... Une fois la partie terminée, elle est archivée et le temps effectué pour réussir le donjon est enregistré.

Pour ce qui est de la discussion en ligne, nous utilisons le protocole Internet Relay Chat (IRC). Ce choix est motivé par la robustesse de ce protocole, éprouvée au fil des années. Mais aussi, par sa grande popularité, qui fait que beaucoup de librairies et implémentations de ce protocole existent dans de nombreux langages, ainsi qu'un petit attachement sentimental pour ce protocole vieux de maintenant trente-deux ans.

Pour ce qui est de la base de données utilisée par le serveur et le site web, nous avons fait le choix d'utiliser RethinkDB, une base de données NoSQL. Nous avons été attirés par la flexibilité que cela permet pour la gestion des documents, par exemple, la possibilité d'avoir une structure unique pour chacun de ceux-ci. Mais aussi, par sa technologie de "sharding" qui permet de facilement mettre à l'échelle les serveurs. Et enfin, par l'utilisation du JSON pour dialoguer avec le serveur, géré nativement par le JavaScript, langage que nous utiliserons pour créer le serveur web.

Le réseau est géré par un serveur, simplement nommé Tesseract-Online. Celui-ci est composé de divers modules :

- RethinkDB : Il est chargé de réaliser la connexion à la base de donnée RethinkDB, et comporete pour l'instant deux fonctions, une permettant d'authentifier une personne avec son pseudo/mot de passe, et une permettant de récupérer un objet UserDTO (Data Transfer Object) à partir d'un pseudo. Cet objet comporte toutes les informations utiles d'un jour, comme son adresse IP si il est connecté, son pseudo/mot de passe, son statut, son niveau...

- Config : Des simples classes permettant de récupérer la configuration des modules IRC et Discord.

- Discord : Une petite fonctionnalité qui permet d'avoir les journeaux du serveur en direct sur notre serveur de discussion Discord du projet.

- IRC : Gère le robot IRC, qui s'occupe d'authentifier les joueurs se connectant. Pour se faire, le client aura un robot irc de son coté lui aussi, qui se connectera au serveur et envoiera au robot du serveur "identify [mot de passe]". Si le mot de passe est bon, il rejoint automatiquement quelques canaux généraux de discussion. Les messages envoyés dans ces canaux apparaissent dans le jeu du client.

- UDP : Nous avons choisi le protocole UDP pour le réseau du jeu, car l'ordre des paquets nous importe peu, et préférons privilégier une latence plus faible. Les échanges se font sous forme de texte, le premier mot indiquant à quel commande transmettre l'information. Par exemple, "CONNECT Thomas supermotdepasse" transmet l'information à la commande Connect, qui va vérifier si ces informations sont bien correctes, et si elles le sont, autoriser "Thomas" à faire des commandes qui requèrent d'être connecté, comme par exemple récupérer les différents salles de jeu ou se connecter à l'un d'entre elle. L'ajout de commandes est aisé, il s'agit d'une simple classe qui hérite de la classe Command, et elle recevra alors toutes les informations nécessaires à sa bonne éxecution.

-GameLogic : C'est ici que toute l'organisation des parties est faite. Actuellement nous pouvons créer des salons et les rejoindre. Lorsqu'une personne rejoint un salon, elle rejoint automatiquement le canal IRC correspondant, afin de pouvoir communiquer avec les membres de son équipe. Cette partie est vouée à grandement s'agrandir pour la prochaine soutenance.

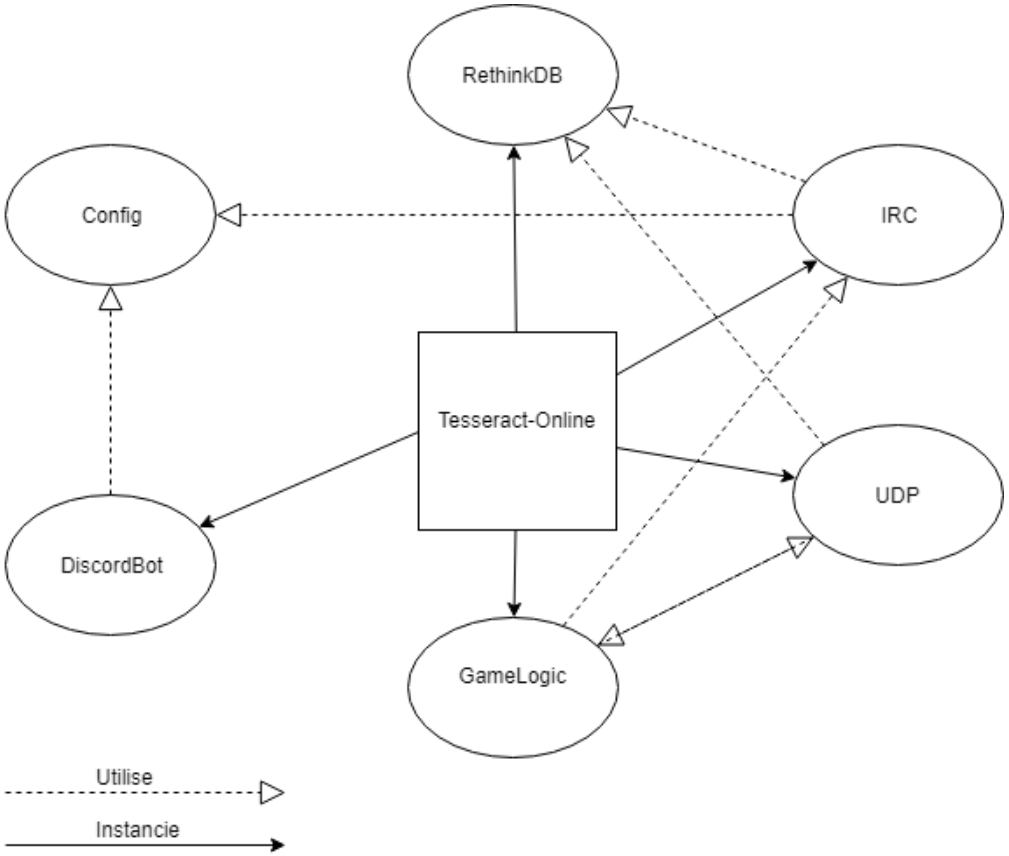


Diagramme des relations existantes entre les différents modules du serveur.

Nous avons ainsi pu réaliser le point d'entrée de notre jeu : la scène Unity de connexion. On y trouve des champs pour y entrer son pseudonyme ainsi que son mot de passe, ainsi qu'une option pour jouer hors-ligne. Dès le démarrage du jeu, le client tente d'envoyer un message "PING" au serveur, et ce n'est qu'à la réponse "PING" de celui-ci, prouvant ainsi qu'il est en ligne et fonctionnel, que le bouton pour se connecter sera accessible. Une fois la connexion acceptée (bon couple de pseudonyme/mot de passe), les objets IRCBot et UDPSocket seront instanciés et accessibles depuis n'importe quel script du projet Unity. Leur utilité est de respectivement interagir avec le serveur IRC et le serveur de jeu.

Le joueur sera ensuite dirigé vers la scène dans laquelle il pourra à loisir rejoindre une salle de jeu, en créer une selon ses paramètres (nom de la salle et graine de génération de la carte de jeu). Toute la partie logique des salons, comme leur création, leur paramètres, les évènements quand un joueur les rejoint/quitte, etc, est géré par le serveur de jeu. Ainsi des commandes ont

été rajoutés afin que le client puisse communiquer ces actions au serveur de jeu. Les échanges se faisant sous la forme de chaînes de caractères, nous avons du inventer un format pour chacune d'elles. Par exemple si le client envoie la chaîne de caractères "LIST" au serveur de jeu, celui-ci lui répondra par plusieurs messages successif les informations sur les différentes salles de jeu disponibles (un message par salle de jeu). Un de ces messages peut ressembler à ceci :

"RINFO #56BEA4 4 Thomas Adrien Eliès William Partie entre amis"

Le client identifie que ce sont les informations d'un salon de jeu grâce au préfixe "RINFO", et traite ensuite cette chaîne de caractères, de sorte à ce que "#56BEA4" soit l'identifiant du salon généré aléatoirement lors de sa création, "4" le nombre de personnes présentes, indiquant ainsi le nombre des mots suivants à prendre pour avoir la liste des joueurs, et le reste de la chaîne de caractères constitue le nom du salon "Partie entre amis". Avoir un identifiant de partie à la fois pour le chat et le jeu permet de standardiser le protocole de communication de Tesseract, tout en ayant un nom de partie complètement personnalisable.

Que ce soit dans la scène des salons de jeu ou en jeu, il existe un "chat" permettant de discuter avec les autres joueurs du jeu ou de sa partie. Accessible avec la touche "Entrer", il est possible de changer le salon textuel dans lequel nous écrivons avec la touche "Tabulation". Ce chat reconnaît les messages privés et les messages dans le salon de discussion de la partie et les affiche dans une autre couleur. Interface et réseau sont ici très entremêlés car chaque évènement reçu par le client IRC va généralement influer directement sur l'interface. Ainsi quand nous rejoignons/quittons un canal de discussion IRC, la liste des canaux dans lequels il est possible d'écrire est mise à jour par exemple. Comme dit précédemment il existe des canaux de discussions pour le salon de jeu dans lequel le joueur est. Ce salon est du nom de l'identifiant de la salle de jeu, expliquant ainsi le "#" contenu au début de celui-ci (les canaux de discussions publiques d'IRC sont précédés d'un "#" pour les différencier des messages privés). Lorsqu'un joueur demande à rejoindre un salon au serveur de jeu, celui-ci va le forcer à rejoindre le canal IRC correspondant, ainsi aucune action n'est requise du joueur, rendant l'expérience de jeu plus agréable.

Réaliser le réseau au niveau du jeu en lui-même nécessite d'agir avec tous les membres du projet, pour faire en sorte que le serveur de jeu puisse ma-

nipuler des joueurs en jeu, les faire attaquer, leur donner un objectif de pathfinding par le réseau... Nous n'avons pas réussi à bien nous coordonner dans la réalisation de cet objectif, et c'est pourquoi il fut notre priorité dès la deuxième soutenance passée.

Nous avons commencé par assigner aux entités du jeu (terme regroupant les joueurs et les ennemis) un identifiant unique. Cet identifiant permet de les désigner lors des interactions entre les clients de jeu. Pour ce qui est des joueurs, chacune de leurs actions est transmise aux autres joueurs sous la forme la plus simple possible. Par exemple, si le joueur 1 décide faire bouger son personnage sur la droite, un message s'enverra aux autres joueurs sous la forme "JINFO 1 102 1" lorsqu'il appuiera sur la touche, puis "JINFO 1 102 0" lorsqu'il la relâchera.

"JINFO" est le code de l'action, "1" le numéro du joueur, "102" le code de mouvement (droite ici) et le dernier "1"/"0" signale on appuie ou relâche la touche. Ainsi, nous pouvons émuler d'autres joueurs dans les parties d'une personne, un peu comme s'il y avait 4 claviers connectés à un ordinateur. Cela a nécessité d'adapter les objets "Player" afin qu'ils puissent recevoir des informations depuis le serveur et non depuis le clavier.

Aussi, toutes les 15 secondes environ, la position de toutes les entitées du jeu du client 1 est envoyé aux autres clients, qui actualisent leurs entitées en fonction de cette liste, ceci afin de prévenir des décalages éventuels pouvant apparaître au cours de la partie.

A la fin de la partie, le serveur envoie le score à la base de donnée.

5.2 Site web

Le site est programmé en JavaScript en utilisant le moteur Node.js. Il s'agit d'une plate-forme logicielle libre et événementielle en JavaScript orientée vers les applications réseau. Ce choix est motivé par l'expérience que nous possédons avec cette plate-forme, promettant un développement rapide.

Lors de la première soutenance, notre site web disponible à l'adresse : http ://tesseract-game.net était basique et ne consistait pour l'instant qu'en une page, présentant les membres du projet, l'avancement ainsi que les différentes ressources utilisées. On pouvait également y apercevoir les pages que nous comptons rajouter dans le futur. Nous n'avions pas besoin de plus à ce moment.

Pour la deuxième soutenance, le site web à donc été "refait" (il n'était constitué que d'une seule page) sous le framework évolutif Vue associé au serveur web Nuxt. Tout le code a été fragmenté en différents composants, rendant le développement et la maintenabilité du site extrêmement aisée.

Ces composants contiennent le code HTML d'une section spécifique, ainsi que du code JavaScript influençant cette section, et éventuellement une feuille de style CSS locale (ne s'appliquant qu'a cette section). Ces composants sont réutilisables dans plusieurs pages différentes, comme le composant "Search", qui permet de rechercher le profil d'un joueur, qui est ainsi utilisé à la fois sur la page d'accueil et le profil d'un joueur, ou encore le composant "NavBar" (la barre de navigation) qui est utilisé sur toutes les pages.

Le Document Object Model (DOM) est une interface de programmation normalisée par le W3C, qui permet à des scripts d'examiner et de modifier le contenu du navigateur web. Par le DOM, la composition d'un document HTML ou XML est représentée sous forme d'un jeu d'objets – lesquels peuvent représenter une fenêtre, une phrase ou un style, par exemple – reliés selon une structure en arbre. À l'aide du DOM, un script peut modifier le document présent dans le navigateur en ajoutant ou en supprimant des nœuds de l'arbre.

Vue est un framework utilisant un DOM virtuel, ce qui permet de ne pas recharger tous les éléments d'une page à partir de la racine à chaque fois qu'un élément est modifié rendant ainsi la navigation fluide est agréable pour la plupart des ordinateurs. Cela permet aussi de manipuler le DOM sans recourir à un modèle HTML, en utilisant du code JavaScript par exemple, facilitant et

rendant plus clair ces opérations dans le code de notre projet. Nous pouvons même faire le rendu du DOM côté serveur puis l'envoyer ensuite au client, minimisant ainsi grandement le travail de celui-ci.

Nous avons donc ainsi créé une page permettant de créer un compte. Les seules informations demandées sont un pseudonyme et un mot de passe, informations que nous envoyons ensuite à l'API REST, laquelle sera détaillée plus tard. Ce mot de passe sera ensuite stocké sous forme salée (une chaîne de caractères connue de nous seuls y est concaténée) et hachée (par l'algorithme SHA-256) dans la base de données pour plus de sécurité. Il va sans dire que si jamais le pseudonyme était déjà pris par un autre joueur, le site en avertirait l'utilisateur.

Nous avons aussi créé une page pour les profils des joueurs. Pour l'instant elle ne contient que le "rang" du joueur (Administrateur, Modérateur, etc..) ainsi qu'un indicateur montrant si cette personne est en ligne dans le jeu ou non. Chaque joueur se voit attribuer une image de profil générée aléatoirement. Pour récupérer ces informations, le site interroge l'API REST sur l'utilisateur en question. Bien que la page d'information ne contiennent que peu d'informations, il seras extrêmement aisé d'en ajouté une fois que le jeu sera suffisamment avancé pour récupérer le résultat des parties en multijoueur par exemple.

Passons maintenant à l'API REST. Une API, (Application Programming Interface), est la partie d'un programme que l'on va exposer au monde extérieur. Cette dernière permet d'entrer des données et de récupérer la sortie d'un traitement. Mettre en place une API permet de séparer les responsabilités entre le client et le serveur, améliorant ainsi l'évolutivité de l'application. Uune API "REST" (Representational state transfer) utilisé généralement le protocole HTTP, et est sans état, c'est à dire que les communications entre le client et le serveur ne dépendent d'aucun contexte provenant du serveur, ce qui permet si le besoin se fait sentir d'utiliser plusieurs serveurs différents pour répartir la charge de trafic des différentes requêtes.

Notre API est faite, comme pour notre site, sous Node.js (une plateforme logicielle libre et événementielle en JavaScript). Nous avons utilisé le framework Hapi.js pour le serveur web, ce qui nous permet de gérer les différentes routes très facilement. Chaque route pointe vers une function, qui va en fonction de la requête (ajout de joueur, demande de profil..) communiquer avec la base de donnée RethinkDB, et retourner les informations sous forme de

document JSON (JavaScript Object Notation). Ce document sera ensuite manipulé par les composants qui en ont besoin dans Vue afin de faire le rendu la page web.

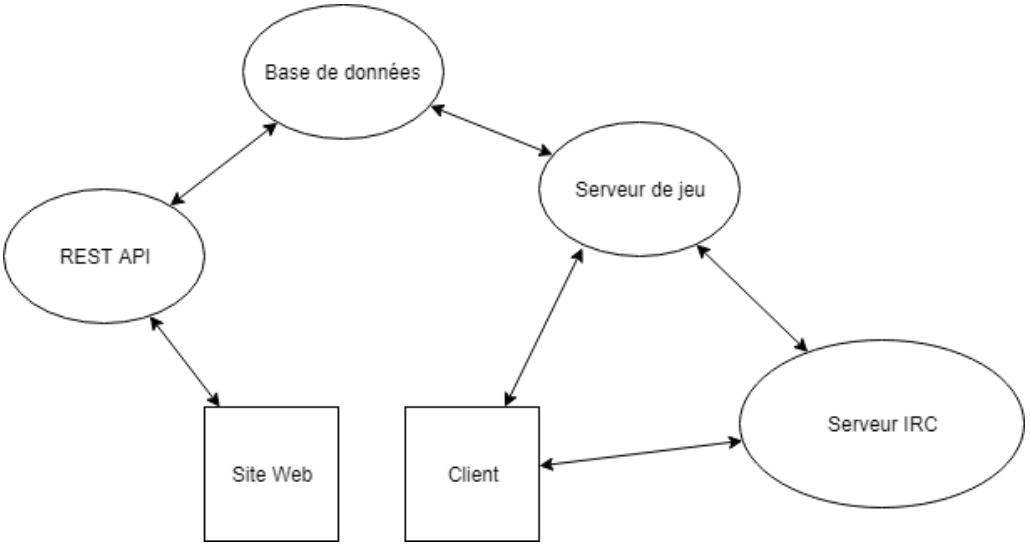


Diagramme des relations entre les différents composants de Tesseract

Enfin, pour la soutenance finale nous avons rajouté un historique des parties sur les pages de profil, ainsi qu'un wiki pour le jeu. Il s'agissait de pages HTML classiques avec du texte, rien de très complexe de ce coté là

5.3 Textures

L'aspect visuel est très important dans n'importe quel jeu. Étant le premier contact avec le joueur, il lui permet de comprendre l'univers dans lequel il va être plongé. Par ailleurs, c'est en partie le visuel d'un jeu qui va déterminer si l'utilisateur est prêt à y jouer. Il est donc primordial de traiter cette partie à part entière afin de pouvoir proposer une expérience de jeu des plus agréable. Nous avons comme idée de créer des textures du type "pixel-art" pour **Tesseract**. L'avantage de ce choix graphique est qu'il nous permet d'apporter un côté "fantasy" à notre jeu. Nous souhaitons que notre carte soit colorée et attirante, afin que le joueur soit directement plongé dans une ambiance unique.

Contrairement aux idées reçues, ce style de graphismes peut donner de beaux jeux. De nombreux jeux récents l'ont confirmé, comme **The Binding of Isaac : Afterbirth** et **Hyper Light Drifer**. Nous espérons faire un rendu dont nous serons fiers, afin de mettre une fois de plus en avant les possibilités artistiques de ce style.

Nous avons commencé par la création des textures nécessaires pour la réalisation d'une salle, afin de satisfaire l'ambiance du jeu. Après quelques réflexions, nous avions convenu de créer les sprites - les images servant des personnages et décors du jeu - dans un style "pixel-art", par hommage à ce style vintage et moteur dans l'histoire du jeu vidéo. Le pixel-art, comme son nom l'indique, consiste à reproduire des objets, personnages ou autre élément de la vie quotidienne simplement à l'aide de pixels. Cet art est notamment présent dans les jeux rétros, comme Donkey Kong ou le célèbre Pac-Man, et très prisé dans les jeux Rogue-Like. Concernant la résolution, nous avons vite conclut de travailler sur du 64x64 pixels, rendant les décors plus détaillés.

Dans un premier temps, notre objectif a été de créer des murs craquelés, servant à délimiter chacune des salles et couloirs. Une dizaine de murs différents nous étaient utiles au commencement de notre jeu pour aborder tous les cas engendrés par la génération aléatoire de la map. De nombreuses textures de mur étaient nécessaires pour la réalisation de chaque mur d'une salle ainsi que chaque mur des couloirs, qui ne sont pas toujours les mêmes. En plus de la difficulté du dessin, des problèmes de perspectives ne sont pas venus nous faciliter la tâche.

Dans un deuxième temps, nous avons créé le sol. Au départ, nous avions une simple texture de sol simple, d'une couleur unie beige - pour ne pas dire

jaune moutarde - mais l'ambition d'en créer une beaucoup plus convenable était déjà dans nos esprits, histoire d'éviter un côté répétitif des salles. L'idée était de présenter notre jeu de manière simple, son principe, sans tenir compte des détails trop importants. Néanmoins, nous avons tenu à rendre les salles un minimum "vivante". Nous avons ajouté un effet d'ombre sur le sol, le but étant de respecter une certaine logique de perspective du donjon. Quelques éléments étaient présents, comme des tables, des ossements ou encore des traces de boues sur le terrain.

Les dernières textures sont celles du joueur et de l'ennemi. Comme spécifié dans notre Cahier des Charges, notre jeu contient 4 classes de personnages jouables ; le Guerrier, un chevalier lourd et robuste, l'Archer, un personnage fragile au corps à corps mais efficace à longue distance, le Mage, un personnage utile contre des hordes d'ennemis et l'Assassin, une classe fragile mais efficace en toute situation. Chaque personnage peut se déplacer dans 8 directions. Pour autant, nous avons choisi de nous limiter à la création et l'animation des textures pour seulement 4 directions.

C'est cette dernière classe - l'Assassin - qui aura eu le mérite d'être notre premier personnage jouable. En plus d'avoir implémenté ses animations, nous avons également crée son attaque de base ; un lancer de shuriken, une texture animée afin de permettre au joueur d'avoir un rendu visuel des attaques lancées par l'assassin.

Enfin, l'ennemi n'était pour l'instant constitué d'une seule sprite, un simple barbare. Pour cette première soutenance nous n'avions besoin que d'un seul type d'ennemi pour montrer le commencement de notre jeu et la possibilité d'interaction entre le joueur et l'ennemi.

Pour la soutenance 2, il nous a fallu reconsidérer les décors et les différents éléments du donjon. Initialement en résolution 64 x 64 pixels, les murs, sols et tout autre élément d'environnement ont été reconstruits sur une base 32 x 32 pixels. Ce changement de taille s'explique par une facilité de gestion des textures lors de la génération aléatoire de la carte. Nous avons dû designer à nouveau les murs mais le résultat n'en est que meilleur selon nous aujourd'hui.

En effet, ayant pris de l'expérience dans le domaine du pixel-art, nous avons pu avoir une meilleure prise de conscience des caractéristiques qui rendent "réelles" les sprites. Ainsi, jeux d'ombres, couleurs et perspectives ont été grandement améliorés et rendent le donjon plus palpable. Ce "redesign" des

différents murs nous a permis de réduire également le nombre de textures nécessaires pour créer des salles, passant de 15 à 7 sprites pour traiter l'ensemble des cas possibles lors de la formation des murs des salles ou des différents piliers.

En plus de ces changements utilitaires, de nouvelles textures ont été ajoutées. Un sol plus détaillé, des pièges infligeant des dégâts au joueur, des flaques, des torches, portails et coffres rendent l'atmosphère du donjon plus sombre et immersive. Cette immersion est par ailleurs renforcée grâce - une fois de plus - aux différents jeux de lumières actifs sur la carte, renforçant l'ambiance mystérieuse de Tesseract.

Concernant les personnages, il a fallu créer les trois autres classes restantes après l'Assassin : le Mage, l'Archer et le Guerrier. Chacun de ces personnages possède un design unique et des animations propres à leur style. Par exemple, chaque classe possède un dash différent ou encore des mouvements d'attaques définissant leur style, avec un coup d'épée pour le Guerrier ou un lancer de boules de feu pour le Mage.

Les textures des personnages sont restées en 64 x 64 pour un souci de précision ; en 64 x 64, les personnages sont plus détaillés et leurs animations les rendent plus "vivants". Les inspirations concernant leur création sont diverses et variées, allant du personnage de Link dans The Legend of Zelda pour l'archer aux armures médiévale classiques pour le Guerrier, en passant par une référence à Black Mage de Final Fantasy IX pour la classe Mage.

Enfin, les ennemis n'ont pas été retravaillés pour cette soutenance, faute de priorité sur d'autres points du projet. Un nouvel type d'ennemi a néanmoins été implémenté, l'Archer Squelette. Son design restait néanmoins temporaire car trop peu travaillé à notre goût.

Pour la dernière soutenance, l'objectif de cette partie a été d'améliorer le travail entrepris lors de la deuxième soutenance. Le plus gros du travail a été de dessiner les ennemis. En effet, n'étant que très développé depuis le début du développement, priorité donnée à la salle et aux personnages, les ennemis héritent maintenant de vraies sprite travaillées et animées. En plus, des ennemis classiques ont aussi été désignés un boss de fin. Le travail a en revanche été mineur sur les personnages et les décors, bien travaillé dans le passé. Ces deux domaines étaient donc passables de retouches, bien que certaines textures aient été ajoutées pour le décor.

5.4 Interface

L'ATH du joueur, autrement dit, les informations disponibles sur son écran, seront des plus efficaces. Le but ici n'est pas de noyer l'utilisateur dans une pluie de renseignements mais de lui afficher ce qui est nécessaire pour qu'il arrive à mener son aventure dans le meilleur des conforts. Dès sa connexion au jeu, l'utilisateur aura accès à un menu détaillant les différents modes de jeux disponibles ; le mode en ligne ou le mode local. Une fois le mode et son avatar sélectionnés, le joueur pourra se lancer dans l'exploration du donjon. En plus de voir la pièce dans laquelle il se trouve, l'utilisateur pourra voir sa barre de santé et accéder à un autre menu. Dans ce dernier se trouvera les divers objets et potions ainsi que l'arbre de compétence grâce auquel le héros du joueur pourra apprendre de nouveaux talents.

En plus des différents menus et informations, des interactions seront disponibles entre l'utilisateur et le jeu. En effet, le joueur pourra converser avec Xelia, une entité mystérieuse, dont seule la voix est perçue. Des dialogues s'afficheront à l'écran et permettront au joueur de suivre l'histoire intrigante du lieu dans lequel il se trouve.

Lors de la première soutenance, l'interface du jeu se voulait des plus fonctionnelles. Au design simple, elle se constituait du logo du jeu ainsi que de trois boutons avec lesquels le joueur pouvait interagir pour effectuer différentes actions. Le bouton Start permettait au joueur de charger la première scène du jeu. Le design et les interactions possibles sur ce menu ont évidemment grandement évolué au fur et à mesure de l'avancement du projet.

Pour cette première soutenance, l'interface avec laquelle le joueur peut interagir se résume au menu de lancement du jeu. Ce dernier évoluera bien entendu au fur et à mesure de l'avancée de notre projet. La page générée est le résultat d'un canevas sur lequel sont superposés différents objets : les boutons, les images et le texte. L'utilisateur a le choix entre trois boutons pour le moment, chacun d'entre eux débouchant sur une action différente. Le bouton "Start" sert naturellement à lancer le jeu. De la même manière, le bouton "Quitter" est relié à l'action "quitter le jeu". Enfin, le bouton "multiplayer" n'est pas fonctionnel pour le moment, mais sa présence témoigne de notre avancée sur le mode multijoueur du jeu, qui vous sera présenté à part lors de notre présentation. Dans le futur, des améliorations seront à prévoir. Par exemple, en appuyant sur le bouton "Start", le joueur pourra choisir sa classe - une fois celles-ci implémentées - avant de partir à l'exploration du Donjon.

Il faudra également songer à la création d'un menu directement accessible lors d'une partie.

Ce futur menu donnera notamment l'accès à différentes interactions. Pour le moment, nous n'avons pas encore idée avec précision de quelles options il sera doté. Nous songeons cependant à l'accès à l'arbre de compétence du personnage. Cet arbre variera en fonction de la classe jouée par le joueur. Il permettra, en échanges de points d'expérience ou d'une autre ressource, de débloquer des capacités spéciales pour le type de personnage joué. Par exemple, si le joueur utilise l'Assassin, il pourra - imaginons - débloquer un dash-attack là où la classe Mage, pour un même nombre de ressources, aura accès à une attaque infligeant des dégâts sur la durée.
Cet arbre de compétence aura donc sa place dans ce menu accessible directement depuis la partie et, de façon générale, une place importante au sein du jeu.

Ainsi, nous continuons à réfléchir aux différentes possibilités qu'offrira ce menu. Nous songeons notamment à implémenter : un bouton "paramètres", afin que le joueur puisse personnaliser entièrement ces touches et jouer à son aise, et un bouton "quitter". Pour finir, l'interface du jeu sera embellie dans le futur avec l'arrivée d'un ATH permettant de visualiser sa barre de vie, son inventaire ou encore d'autres informations utiles pour le joueur.

Pour la seconde soutenance, une première amélioration a été le système de login en guise de menu. Le joueur pouvait choisir de jouer en ligne. Cependant, le joueur avait toujours le choix de jouer en "offline", hors ligne où il sera seul dans le donjon contre les monstres.

En jeu, de nouvelles informations sont disponibles à l'écran, permettant notamment au joueur de visualiser son état à tout moment dans la partie. Ainsi, trois barres sont disponibles : une barre de santé, une barre d'énergie et une barre d'expérience. Ces trois informations se mettent constamment à jour ; si le joueur subit dégâts ou gagne de la santé, la barre de vie sera actualisée en temps réel.

Enfin, l'inventaire du joueur a été affiché mais n'était pas encore fonctionnel.

Pour la troisième soutenance, le travail pour l'interface s'est divisé en deux parties : d'abord des ajouts nécessaires, ensuite des ajouts mineurs. Parmi

les ajouts nécessaires, on retrouve le menu intermédiaire. Ce menu permet au joueur de mettre en pause le jeu et, à tout moment, de quitter la partie en cours. Ce menu n'est rien de plus qu'un canvas qui, une fois activé, appelle un script mettant en pause le jeu. Si le joueur quitte la partie, il sera directement renvoyé au menu d'accueil.

Par la suite, la seconde interface ajoutée a été l'affichage de la mort du joueur. Pour se faire, une nouvelle scène a été créée, directement activée lors de la défaite du joueur. Une fois dans cette nouvelle, le fameux "GAME OVER" s'affiche, accompagné d'un message aléatoire pour motiver le joueur - ou non - à poursuivre l'aventure. De même que pour le menu intermédiaire, le bouton quitter présent dans cette scène renvoie directement au menu.

En plus de ces ajouts, des modifications ont été portées sur l'ATH in-game. L'inventaire, par exemple, est désormais fonctionnel. Le joueur peut donc voir quelles potions il possède et laquelle utiliser. Les autres modifications concernant l'ATH sont esthétiques. Par exemple, des ajouts visuels mineurs - mais indispensables - comme par exemple la possibilité de voir la barre de vie des ennemis ou encore de voir la touche d'interaction avec le coffre.

5.5 Mécanique de base

Afin de commencer la création du joueur et de ses mécaniques de jeu, nous devons comprendre ce qui le constitue. Un personnage requiert plusieurs variables, certaines ne sont que de simples entiers, comme la vie, l'expérience ou les dégâts. Cependant, certaines sont plus compliquées à stocker, notamment les animations sous la forme d'AnimationClip ainsi que les textures ou encore les compétences du joueur. Il fallait donc trouver un outil capable de stocker aisément toutes ses données.

La solution utilisée est l'utilisation des Objets Scriptables. Un objet scriptable est défini comme un "data container" d'après Unity, c'est-à-dire un outil permettant de stocker des données, il nous permet notamment d'avoir une instance d'une classe sous la forme d'un Asset, nous permettant de modifier ses variables à volonté et très aisément.

L'objet scriptable nous permet donc de stocker facilement ces variables, en créer une instance et les modifier depuis l'inspecteur d'Unity. L'utilisation de ce type de classe nous permet donc de simplifier grandement la manière de stocker les données pour nos quatre classes.

Nous avons maintenant un moyen de stocker ce qui concerne le joueur, nous devons commencer à créer les déplacements de celui-ci ainsi que ses collisions. Certains problèmes ont été rencontrés, notamment quand le joueur se déplace rapidement ou quand la zone de collision de l'objet est très fine. Dans ces conditions, il n'est pas rare de voir le joueur traverser un mur, rebondir légèrement dessus ou encore rester coincé. Il nous a donc fallu trouver un outil capable de résoudre ce léger problème, nous avons choisi d'utiliser un système de "Raycast". La solution employée a de nombreux avantages, elle permet de régler les problèmes de collisions qui n'étaient pas gérés automatiquement par Unity, puisque la vitesse du joueur n'a aucun impact sur le "Raycast" de plus peu importe la largeur de la zone de collisions, du moment qu'elle existe, le "Raycast" sera en mesure de la repérer. Un autre avantage du "Raycast" est la flexibilité dans les déplacements du joueur, celui-ci peut se déplacer indépendamment des collisions rencontrées sur les axes X et Y, une option que le système intégré dans Unity ne nous permettait pas causant certains problèmes lorsque le joueur se collait dans un mur et essayer de faire des mouvements dans une diagonale.

Nous avons également implémenté un "Dash", c'est-à-dire une capacité de déplacement rapide, il permet au joueur de se téléporter sur une courte

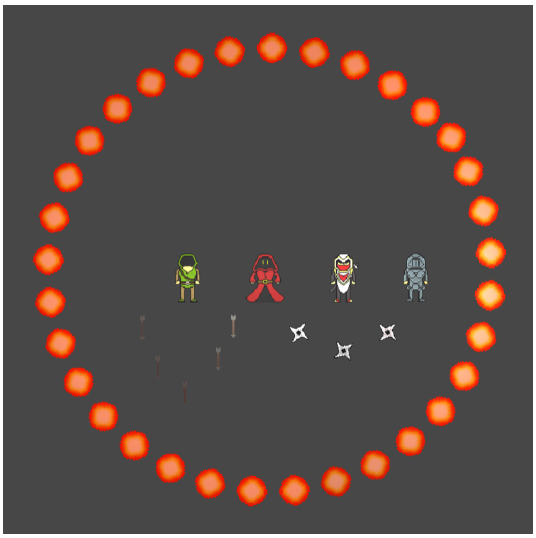
distance en fonction de la position de son curseur. Nous avons également vérifié que le joueur ne puisse pas passer au travers d'un mur.

Notre joueur est maintenant capable de se déplacer, mais il est un peu rigide dans ses déplacements, en effet sa texture ne change pas, un système d'animation s'impose donc pour notre personnage. En l'occurrence, nous avons besoin de quatre animations de déplacement, d'attente, d'attaque et de dash. Pour autant, en fonction du choix du personnage, la texture utilisée pour l'animation ne doit pas être la même. Les outils proposés par Unity sont très efficaces mais très peu malléable, nous avons donc préféré passer par des scripts pour gérer cela. C'était donc beaucoup plus complexe, mais cela nous a permis d'avoir un système qui fonctionne peu importe le nombre de personnage que nous possédons.

Notre personnage se déplace librement, mais ne peut pas attaquer, c'est donc notre prochaine étape : implémenté une attaque de base. L'assassin peut envoyer des shurikens, le mage des boules de feu, l'archer des flèches et le guerrier peut attaquer avec son épée. Il nous fallait donc créer un système de vie pour les ennemies afin de pouvoir les tuer. Les animations du joueur et des projectiles lancés par celui-ci devaient également être implémentés dans le jeu. Nous avons également ajouté des compétences spéciales disponibles sur chaque personnage, elles ont pour but de donner un style de jeu différent à chaque personnage voici un exemple pour chaque personnage celle-ci : L'assassin peut devenir invisible et faire plus de dégâts sur sa prochaine attaque de base, le mage peut tirer des boules de feu tout autour de lui, l'archer peut augmenter temporairement sa vitesse d'attaque pour tirer beaucoup plus rapidement, le guerrier peut gagner un bonus de vie sur une courte durée afin de survivre plus longtemps aux ennemies.

Le personnage doit également évoluer. Il a donc fallu créer un système de niveau et d'expérience. Il fallait également gérer ce système indépendamment du personnage choisi, bien que le système était globalement le même, cela a entraîné quelques complications. Notre personnage peut donc gagner de l'expérience en tuant des monstres. Cela lui permet de monter de niveau afin d'améliorer ses statistiques. Bien évidemment, afin de permettre à chaque joueur de varier son style de jeu il est possible d'améliorer certaines compétences du personnage jouer grâce aux points de compétence débloquer à chaque niveau gagné.

Actuellement, nous avons donc 4 classes jouables. Elles peuvent se déplacer, attaquer et utiliser des compétences, monter en niveau ainsi que s'améliorer. La prochaine étape est donc de faire interagir le personnage avec la carte et ce qui la compose. La première chose était de trouver un moyen de lui faire ouvrir un coffre et de récupérer ou non son contenu en fonction de ce que le joueur souhaite faire. Il doit également être capable de récupérer les objets laissés par les ennemis tuées.



Rendu visuel de nos quatre personnage utilisant une de leur compétence

Ce qui nous amène à la création d'un inventaire afin qu'il puisse stocker les objets qu'il trouve pendant sa partie. Actuellement, l'inventaire lui permet de stocker une arme et 4 potions. Il est également possible pour le joueur d'utiliser ses potions s'il le souhaite.

La dernière étape était de sauvegarder les informations de notre joueur, c'est-à-dire toutes ses caractéristiques et son inventaire. Pour éviter que le joueur puisse modifier aisément les informations de son joueur en modifiant directement les fichiers de sauvegardes, nous utilisons une classe spéciale pour chiffrer des informations sous la forme d'un fichier binaire. Cette partie s'avère assez complexe, puisque chaque statistiques évolue différemment, il fallait toute les stocker, le plus problématique était les compétence du joueur, en effet une compétence n'est pas un type primitif, cela implique qu'il ne peut être stocké tel quelle, il a donc fallu trouver un moyen de permettre au joueur de récupérer les bonnes améliorations sur chacune de ses compétences, et cela, peut importe le personnage qu'il a choisis de jouer.

Bien que cela ne se rapproche que peu des mécaniques de bases, il fallait un moyen pour faire communiquer les éléments entre eux, indiquer aux ennemies quand ils subissent des dégâts, indiquer à l'ath que le joueur a perdu de la vie et ce genre d'action.

Il était nécessaire de trouver un moyen de communiquer entre tous les scripts de tous les objets. Cependant, pour garder un code propre et facile à déboguer, il ne fallait pas directement accéder aux composants que l'on souhaitait modifier. En effet, en supposant que pour que l'ath change la barre du vie du joueur en fonction de ses points de vies, celle-ci accède directement au joueur, cela ne posera pas de problèmes la plupart du temps, mais quand le joueur va mourir alors il sera impossible d'accéder au joueur et le jeu va donc complètement planter.

Pour faire face à ce problème Unity propose un système de "GameEvent". Cela a pour effet de permettre de dire à n'importe quelle fonction de n'importe quel objet qu'une certaine action a eu lieu. Il indique donc uniquement qu'un évènement précis est apparu, il n'y a donc aucune dépendance entre les objets.

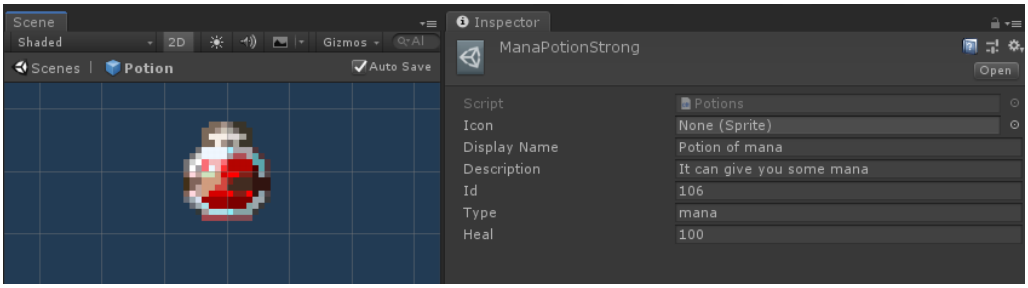
Nous avons d'abord créé un système de "GameEvent" en 2 parties, une qui énonce un changement et une qui écoute ce changement. Imaginons que l'ennemi attaque le joueur, nous pouvons créer un évènement "PlayerGetDamage" qui s'activerait quand le joueur subit des dégâts. Ainsi, dans le script attaché au joueur nous pouvons créer un évènement qui écoute l'évènement "PlayerGetDamage" et qui s'activera en même temps que celui-ci. Ce système est très intéressant, car un "GameEvent" permet d'interagir avec autant de "GameEventListerner" que nous le voulons. Mais ce système est tout nouveau et malheureusement que très peut développé, il nous est impossible en l'état d'envoyer une information sous n'importe quelle forme, elle doit être d'un type bien précis et unique.

Il fallait donc également améliorer ce système afin de pouvoir envoyer n'importe quelle information. Pour cela nous avons dû créer une structure de donnée qui s'occuperait de stocker les informations que l'on souhaite, il fallait que toutes ses structures soient du même type avec que les GameEvent d'Unity sois capable de fonctionner, cela était un peu perturbant à utiliser mais cela c'est avérer extrêmement pratique. Puisque nous pouvons communiquer aisément avec tout les objets du jeu sans risque de le faire planter à tout instant.

5.6 Objets

Le travail sur les objets s'est réparti sur les deux dernières soutenances. En effet, pour la première soutenance, nous devions d'abord avancer sur d'autres parties essentielles du projet avant de pouvoir travailler de manière efficace sur les objets.

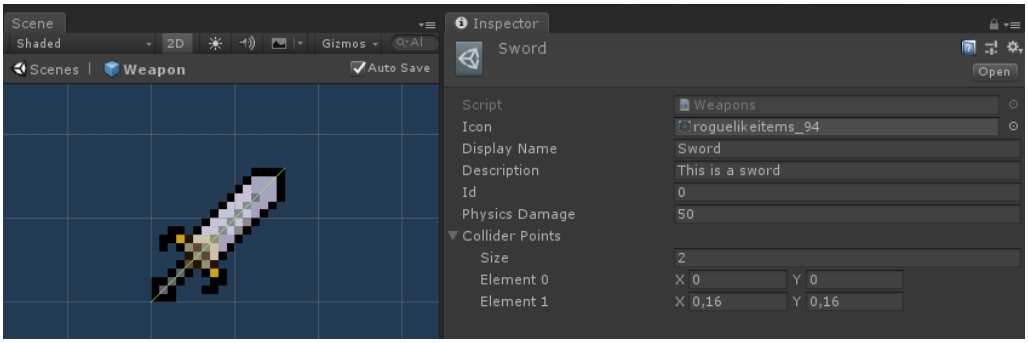
Dans un premier temps, nous avons mis en place les objets activables, c'est-à-dire les coffres et les potions. Les potions sont des consommables, cela signifie qu'une fois utilisée la potion est détruite. Actuellement, il y a deux types de potion, celle de soin et celle de mana, réunies sous un seul objet scriptable. Elles permettent au joueur de régénérer sa caractéristique correspondante, vie ou mana, en consommant la potion. Chacune est déclinée en trois niveaux de rareté, Small / Medium / Strong, qui régénère plus ou moins. Pour créer ces six potions, il n'a fallu que quelques clics, une fois l'objet scriptable implémenté. Celui-ci stocke les informations concernant la potion, il suffit de modifier le type de la potion et la valeur de régénération pour en créer une nouvelle. Il s'agit là d'un très bon exemple de l'utilité des objets scriptables. En effet, malgré leur prise en main assez complexe, le temps et l'énergie nécessaires à leur mise en place est récupéré dans la création d'objets comme les potions. Ensuite, il est possible d'obtenir les potions en ouvrant des coffres. Chaque coffre confère au joueur une nouvelle potion aléatoire si son inventaire n'est pas déjà plein. Les coffres ne sont pas détruits une fois utilisés mais ils sont eux aussi activables une unique fois, une fois ouverts ils ne peuvent plus fournir de ressource au joueur.



Potion : Texture / Objet Scriptable

Quant aux armes, lors de la deuxième soutenance, elles n'étaient pas encore fonctionnelles mais nous avions déjà commencé leur implémentation. Notre objectif n'était pas d'avoir une seule arme fonctionnelle mais d'avoir une base d'arme malléable pour pouvoir créer différentes armes aussi simplement que l'on peut le faire pour les potions. Nous avons donc commencé par créer la Prefab et l'objet scriptable de celles-ci. Ce dernier contient notamment la

texture de l'arme et les informations sur les vecteurs nécessaires à la création de son composant EdgeCollider. Celui-ci permet de déterminer si l'arme est entrée en contact avec un ennemi lors de l'attaque afin de retirer des points de vie de l'ennemi la valeur de dégât de l'objet. Malheureusement, ceci n'est pas encore implémenté car nous avons rencontré un problème concernant l'animation de l'arme. Toujours dans une optique de simplification de la création d'arme, nous avons tenté d'animer l'arme sans avoir à créer des textures supplémentaires et sans passer par le système d'animation d'Unity. Cela s'est d'abord avéré être un échec car le rendu était très approximatif et ne nous convenait pas.



Epée : Texture / Objet Scriptable

Dans un second temps, nous avons donc commencé par régler le problème d'affichage qui ne nous convenait pas. Il fallait trouver une méthode à la fois adaptée pour infliger des dégâts à l'ennemi en modifiant au minimum notre implémentation. En effet, nous ne souhaitions pas perdre de vu notre premier objectif, celui d'une implémentation la plus malléable, permettant de créer les nouvelles armes sans consacrer un temps démesuré à la création de textures et d'animations. Nous avons finalement atteint cet objectif.

Pour cela, nous avons eu recours aux évènements, ou GameEvent. Il s'agit d'un système mis à disposition par Unity et précédemment adapté pour son intérêt pour les mécaniques de base. Ici, les évènements sont utiles afin de déplacer l'arme, d'abord en fonction des déplacements du joueur, puis lorsqu'il attaque.

Ensuite, nous avons aussi utilisé le système de Coroutine. De manière générale, lorsqu'une fonction est appelée, elle est exécutée d'une seule traite avant de passer passer à prochaine "frame", c'est-à-dire à la prochaine image à afficher. Cela pose problème lorsqu'on souhaite modifier une valeur de manière graduelle mais ce à partir d'un seul appel de la fonction. Une coroutine

permet de passer outre cette limite. En effet, elle permet de suspendre l'exécution de la fonction jusqu'au calcul de la prochaine image. Dans notre cas, cela permet de modifier la position de l'arme plusieurs fois après le déclenchement de l'attaque, afin d'obtenir un mouvement relativement fluide.

Ensuite, une fois les armes fonctionnelles, nous avons ajouté leur génération à la mort de l'ennemi. Ce ne fut pas particulièrement complexe. Pour effectuer cette tâche, nous sélectionnons aléatoirement une arme parmi la liste de celles-ci lors de la création de l'ennemi. A la mort de celui-ci, nous instancions l'arme à sa position courante, avant de le détruire. Suite à cela, le joueur peut récupérer l'arme grâce au système implémenté dans les mécaniques de base.

Enfin, nous avons achevé notre travail sur les objets en créant quelques armes possédant différentes raretés et donc différentes caractéristiques. Encore une fois, les objets scriptables nous ont permis d'effectuer cette tâche bien plus simplement que si nous avions du créer une "prefab" pour chaque arme.

5.7 Carte

Nous souhaitions avoir un carte qui ressemble à un donjon, elle doit contenir plusieurs salles d'une taille aléatoire ainsi que des couloirs pour les relier, nous ne voulions pas que cela sois trop labyrinthique, de ce fait nous avons choisis de générer principalement de grosses salles avec le moins de couloirs possible pour les relier.

Afin de générer les éléments de la carte nous avons besoin de la représenter à l'aide de certains outils, nous avons simplement choisis d'utiliser une matrice de la bonne taille, toutes les cellules d'une salle sont assignées à un objet correspondant au sol. Les cellules les plus au bords sont assignées a des murs en fonction de leurs position, de cette manière nous pouvons générer les 4 murs pour chaque direction ainsi que les 4 murs qui se trouvent dans les coins.

Le premier travail fut donc de concevoir un script permettant la génération d'une salle simple, c'est-à-dire uniquement le sol et les murs. Chaque salle est rectangulaire, elle possède une hauteur et une largeur générées aléatoirement entre un maximum et un minimum configurable. Les salles ne possède pas de portes, elles sont donc ouvertes aux autres.

Ensuite il faut relier les salles entre elles, nous avons donc un script qui est séparé en deux parties, tout d'abord il choisit une salle aléatoire parmi celles déjà présentes sur notre carte. Il choisit ensuite un point cardinal aléatoire. Celui-ci servira de direction pour créer la prochaine salle, ainsi que le couloir pour la relier à la précédente. Il vérifie tout d'abord qu'il est possible de créer une salle dans cette direction, c'est-à-dire qu'il n'y a pas déjà une salle, un couloir, ou encore qu'il existe déjà un chemin pour relier ses deux salles entre-elles. Si toutes les conditions sont réunies, il va chercher à créer un couloirs pour les relier.

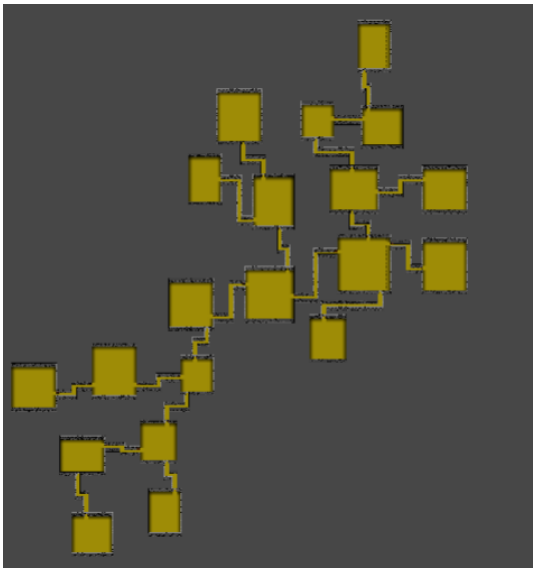
Afin de créer les couloirs reliant les salles, il faut réfléchir aux différents chemins qui existent. Il existe 8 combinaisons de chemins possibles, 2 de plus si l'on compte le cas particulier où les portes sont axées, soit en X soit en Y. Afin de générer un chemin correct et maniable, notamment pour le choix des textures à ajouter sur le chemin, nous séparons celui-ci en 3 routes.
- Dans le cas où le chemin se dirige selon l'axe X, nous avons : le mur du haut, le sol, et le mur du bas.
- Dans le cas où le chemin se dirige selon l'axe Y, nous avons : le mur de droite, le sol et le mur de gauche.

Dans le but de créer les chemins plus aisément, chacune de ces routes comporte 4 points, ces points sont calculés en fonction de la position des salles ainsi que de la direction que le chemin doit prendre. Le premier est situé au niveau de la première salle et le dernier au niveau de la l'autre salle. Les 2 points intermédiaire sont placé au milieu du chemin, l'un axé avec la première salle et l'autre axé avec la seconde salle. Ces 4 points permettent de créer une ligne qui formera la route recherché.

Cela nous permet de choisir plus facilement les textures sur chaque portion de route, ainsi que les textures qui sont situées dans les coins. Le plus gros problème est justement le choix des textures à utiliser pour chacun des murs. En effet, en fonction de la position de la première porte par rapport à la deuxième, ou de la direction empruntée par le chemin, les textures sont radicalement différentes. Un autre détail à prendre en compte est la perspective, une solution à ce problème est de décaler les bons murs de quelques pixels afin de donner l'impression d'avoir un couloir plus fin.

Enfin, il fallait créer les collisions de tous les murs afin de respecter les perspectives et d'empêcher le joueur de quitter le terrain de jeu. Un des problèmes des collisions vient du fait que 2 murs avec les mêmes textures nécessitent parfois des zones de collision différentes, notamment les murs d'angles, qui sont différent pour les couloirs ou pour les salles. Ce qui complique encore la génération des salles et des couloirs. Le principal avantage de cet algorithme est sa malléabilité. La taille de chaque salle, la distance qui les sépare en X ainsi qu'en Y est réglable indépendamment et permettra de créer des salles plus grandes en fonction des niveaux. La génération des couloirs est aussi très paramétrable.

Enfin, chaque salle possède 2 caractéristiques qui se révéleront intéressantes pour la génération des ennemies. Tout d'abord une aire, nous pourrons donc ajuster le type et le nombre d'ennemi présent en fonction de cette caractéristique, ainsi que leur type. Cette caractéristique se révélera encore plus importante en fonction de la classe utilisée par le joueur puisque nous pourrons l'ajuster afin d'ajouter de la difficulté, ou au contraire en retirer. Cela permettra également de choisir les décors que nous pourrons ajouter dans notre salle, certains décors seront imposants et nécessiteront beaucoup de place afin d'être disponible dans la salle. Sa deuxième caractéristique est sa distance par rapport à la salle d'origine. Plus le joueur s'éloigne de l'endroit où il est apparu, plus les ennemis seront puissants et nombreux.



Rendu visuel final de la carte

Malheureusement bien que la carte et le système employée dessus était fonctionnel, il ne nous satisfaisait pas. La carte n'avait pas le rendu souhaiter, de plus ils était parfois assez longs de la générer, elle présentait également d'autres défauts plus ou moins important. Cependant, avant de s'attaquer à un autre algorithme de génération de carte, il fallait d'abord comprendre ce qui ne nous plaisait pas et qu'elles étaient les raisons nous forçant à reprendre à zéro la génération de la carte. Tout d'abord, la génération n'était pas assez aléatoire. Puisque les salles se répandaient autour d'une salle centrale, la carte était trop ronde. Les salles ne pouvaient pas se chevaucher et les couloirs n'avaient pas d'intersection. L'implémentation empêchait également de régler ces problèmes facilement. Il fallait donc trouver une implémentation permettant d'avoir des salles et des couloirs pouvant se superposer et avoir un rendu réellement aléatoire. Il fallait également simplifier l'utilisation de la grille par l'algorithme de création du graphe nécessaire au pathfinding, car celle-ci était très coûteuse en ressources et donc en temps.

J'ai commencé par limiter la zone de génération en créant un tableau de booléen, "True" représente une case sur laquelle se trouve un sol, "False" une case qui n'en contient pas. La première étape est de créer le sol de chaque salle, nous prenons de manière aléatoire une position, une longueur et une largeur sur la grille et nous regardons si une salle ne s'y trouve pas déjà. Nous pouvons ensuite choisir de créer la salle ou non, Nous avons donc un système

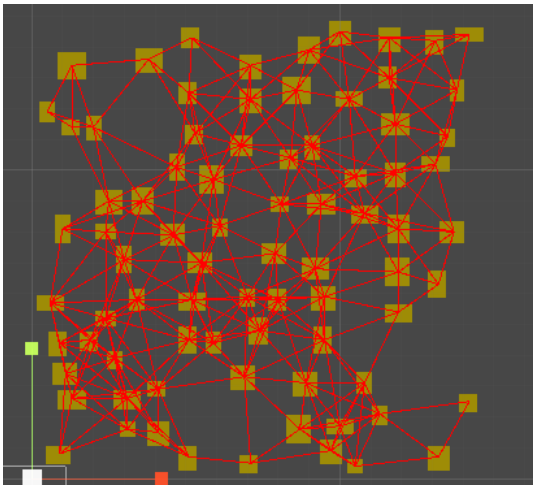
qui permet d'avoir des salles qui fusionnent si nous le souhaitons, chose que l'ancien algorithme de permettait absolument pas. Le système pour créer les salles est donc extrêmement simples puisque nous les créons pratiquement complètement aléatoirement, en conséquence il va être plus complexes de créer les couloirs afin de les relier.

Maintenant que nous avons des salles disposées de manière aléatoire un peu partout sur la carte, nous devons trouver un moyen de les relier sans en oublier une seule. Notre première option était de simplement relier les salles une à une en fonction de leur ordre de création. Mais cela avait pour cause de créer des couloirs beaucoup trop longs et beaucoup trop fréquents. La carte ressemblait presque à un labyrinthe plutôt qu'un donjon. Il fallait donc trouver un autre système pour relier les salles.

Après quelques recherches, Nous avons opté pour un algorithme d'arbre couvrant minimum. Le principe de cet algorithme et de relier tout les noeuds d'un arbre entre eux, en utilisant le moins de chemins possibles et d'avoir des chemins les plus courts possibles. C'était assez proche de ce que nous recherchions. En effet, nous souhaitions relier les salles par des chemins, qu'il y en ait le moins possible et qu'ils soient le plus cours possible. Les salles s'apparentent donc à des noeuds dans un graphe. Nous avaions plusieurs choix pour implémenter cet algorithme, celui de Prim ou celui de Kruskal. Celui de Prim était plus compliqué à implémenter. Mais ce qui nous a permis de trancher entre l'utilisation de ces deux algorithmes, c'est leur complexité. En effet, l'algorithme de Prim est plus optimal quand il y a peu de chemins, mais beaucoup de noeud, à l'inverse, celui de Kruskal est plus adapté pour des graphes avec peu de noeuds et beaucoup de chemins. Dans notre cas, nous avons environ vingt salles, mais pour créer un graphe entre elles nous devons toutes les relier. Pour n salles, nous aurions n^2 liens. Il fallait donc utiliser l'algorithme de Kruskal.

L'algorithme de Kruskal est un algorithme gourmand, c'est-à-dire qu'il va chercher les optimums locaux et non un optimum général. Cela ne nous pose pas de problèmes, nous ne recherchons pas la solution parfaite mais une solution suffisamment bonne pour être valide et cette algorithme nous le permet. Son principe est simple, l'algorithme va trier les chemins des plus courts au plus longs, il va ensuite les traiter un à un et relier les noeuds aux deux extrémités de ce chemin. Cependant, il faut vérifier que relier ces noeuds ne va pas créer une boucle, ce qui serait très problématique. Afin de vérifier cela, il faut créer un ensemble qui contiendra les différents noeuds. Il

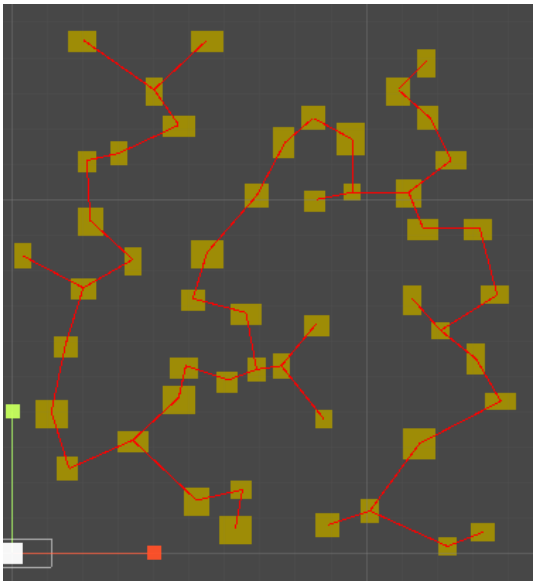
suffira donc de ne pas relier deux noeuds qui appartiennent déjà au même ensemble. Cependant, l'algorithme peut avoir plusieurs milliers de chemins si notre nombre de salles est important. Il fallait donc utiliser une structure optimisée.



Liens possible entre les salles

L'algorithme retenu est nommé "Union-Find". Il permet de créer des liens entre les noeuds de manière optimale et de rechercher si deux noeuds sont déjà reliés d'une manière également optimale. Son principe est assez simple, tous les noeuds sont stockés dans un tableau. Ils possèdent deux caractéristiques, leur parent et leur racine. Quand on souhaite relier deux noeuds il suffit de prendre un noeud comme racine et de définir ce noeud comme le père de l'autre, de cette manière si deux noeuds ont la même racine, ils sont déjà reliés entre eux. Mais ce système risque de créer des liens très longs. D'où l'utilisation d'une fonction permettant de modifier le père en même temps que l'on recherche la racine. De cette façon, quand on recherche la racine d'un noeuds, qui représente son identifiant d'ensemble, on remplace le père de ce noeud par le père de son père, ainsi une fois le noeuds racine trouvé, tout les noeuds par lesquels nous sommes passé auront pour père le noeuds racine, permettant de diminuer le temps de recherches de l'ensemble d'un noeud. L'algorithme permet également de fusionner deux ensembles en ajoutant le plus petit ensemble sur la plus petite branche de l'autre, permettant de créer des liens le plus court possible.

À l'aide de ces deux algorithmes, nous pouvons donc avoir des salles qui sont reliées entre elles sans avoir trop de couloirs placés n'importe comment. Chaque salle et relier à l'aide d'un unique couloirs, cependant à cause de cela aucun couloirs n'était superposé et les salles n'avait que peu de couloirs Il fallait donc rajouter un moyen de créer des couloirs supplémentaires entre les salles pour ne pas avoir des salles qui ne menait à rien. Ceci s'est avéré assez simple. Il suffisait d'accéder à la liste de salles, d'en prendre deux au hasard et de relier leurs centres. L'avantage de faire cela c'est que certains couloirs vont traverser d'autre salle créer donc des liens supplémentaires permettant plus de réalisme.



Lien après l'utilisation de l'algorithme de Kruskal

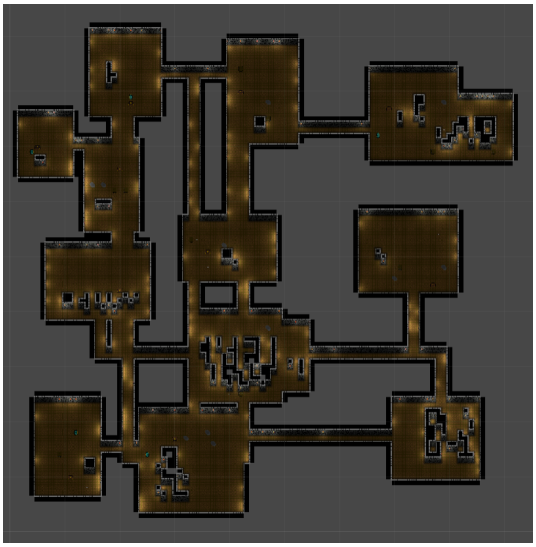
Pour finalisé la génération de la carte, nous devons ajouter une salle pour le Boss, système qui n'était pas présent avec l'ancienne méthode. Pour cela nous ajoutons un portail de téléportation aléatoirement sur la carte qui nous amènera ans une salle différente dans laquelle se trouve le boss. Une fois celui ci vaincu un coffre apparaîtra et le joueur pourra donc récupérer des butins importants avant de continuer vers le prochain niveau.

La prochaine étape était d'ajouter les textures. Pour cela, nous parcourons chaque case de notre tableau et observons ses 8 voisins. Cela nous permet de savoir quel type de mur nous devons placer. Même si cela parait simple,

il existait des milliers de combinaisons possibles, plus précisément quarante mille trois cent vingt pour chaque cellule, pour une carte généreuse cela s'élevait donc à plus d'un milliard de combinaison à calculer. Il n'était donc pas simple de trouver un moyen de toutes les calculer sans partir dans un code trop incompréhensible et surtout trop gourmand en ressource.

Maintenant, nous devons ajouter du décor. Nous avons commencé par ajouter des ombres autour des murs. Le problème était le même que pour la création de mur, il fallait prendre des millions de cas en compte. Le système est donc le même que pour les murs. Plutôt que d'ajouter des grosses structures nous avons préféré ajouter des murs dans les salles de manière aléatoire. Pour cela, il fallait modifier la grille de booléen avant le choix des textures afin d'avoir un résultat qui n'était pas complètement incohérent. Nous avons également ajouté des coffres placés de manière aléatoire dans la salle. Leur contenu est lui aussi aléatoire. Il est possible de modifier facilement le nombre de coffres par salle en modifiant une simple variable. Nous avons également ajouté des portails de téléportation. Ils peuvent déplacer le joueur partout sur la carte. Afin de rajouter un peu de difficulté, des pièges ont été placés sur le sol.

Pour rendre le jeu plus élégant, un système intégral de lumière a été mis en place. Les sources de lumière sont les ennemies, le joueur, les torches et certaines compétences du joueur. Unity nous permet assez simplement d'avoir un rendu élégant de lumière. Mais pour cela, il fallait créer un matériel spécial qui allait s'occuper de modifier l'éclairage des textures en fonction de la lumière ambiante. Il n'était pas aisé de trouver les bonnes valeurs pour ce composant. En effet, en fonction de l'élément, il fallait un système différent. Les ombres devaient être transparentes et passer par-dessus le joueur alors que la plupart des autres éléments devaient simplement recevoir de la lumière. Il fallait également ajouter les lumières sur les torches, les joueurs et les ennemis. Les compétences permettent également de faire de la lumière d'une couleur qui dépend de leur texture.



Nouveau rendu final de la carte

Notre carte est maintenant terminée, cependant même si elle n'est pas un labyrinthe, il est tout de même compliqué de se retrouver dedans, car elle reste très grande. Nous avons donc décidé d'ajouter une mini-carte informative afin que le joueur puisse avoir un rendu plus grand des alentours. Au fur et à mesure que le joueur se déplace sur la carte, la mini-carte va s'agrandir afin de lui permettre de sauvegarder les endroits dans lesquels il s'est déjà déplacé.

L'ajout de cette mini-carte n'était pas aisé, car Unity est très gourmand et utiliser une nouvelle caméra pour la mini-carte était très coûteux. Il a donc fallu changer entièrement l'implémentation de la carte afin de pouvoir l'optimiser grandement. Initialement, chaque case du jeu représente un objet et tous ses composants, quand Unity créé un objet, celui-ci possède des propriétés qui lui permettant d'interagir avec les éléments extérieur, cependant nos sols et nos murs n'ont aucune interaction avec le joueur, outre les collisions.

C'est pour cela que nous avons utilisé un système nommé "TileMap", ce système est une grille de texture permettant de diminuer drastiquement les coûts en performance des objets utilisé par Unity, en effet une texture est simplement une image-là ou un objet Unity possède de nombreuses caractéristiques complexes, notamment pour gérer la physique du jeu.

Afin d'avoir notre mini-carte nous allons donc créer plusieurs "TileMap" et nous ne les afficheront pas tous aux mêmes endroits, d'un coté nous avons la

mini-carte qui sera affiché en haut à gauche de l'écran et de l'autre le terrain qui sera affiché en grand sur tous l'écran du joueur.

5.8 IA

Dans un premier temps, nous avons commencé par poser les bases de notre IA, celle-ci était alors tout à fait naïve. Elle consistait premièrement à calculer le plus court chemin entre deux entités, ici l'ennemi et le joueur, grâce à un algorithme de "pathfinding". Deuxièmement, elle s'occupait de déplacer l'ennemi sur la carte, le long du chemin calculé, jusqu'au joueur.

Avant toute chose, nous nous sommes documentés sur les différents algorithmes de recherche de chemin. En effet, plutôt que de se lancer tête baissée dans une implémentation hasardeuse, il était préférable de déterminer la solution la mieux adaptée à nos besoins. Suite au croisement de plusieurs sources, l'algorithme A* s'est rapidement imposé face à ses homologues. Cet algorithme à de très nombreuses fois fait ses preuves par le passé dans différents jeux vidéo, domaine dans lequel il est particulièrement adapté. Effectivement, il permet de calculer le plus court chemin entre 2 nœuds d'un graphe tout en nécessitant peu de ressources. Cela permet donc de l'utiliser sur plusieurs ennemis en même temps sans demander un effort insurmontable à la machine.

Tout d'abord, une fois l'algorithme de pathfinding choisi, nous avons travaillé à représenter la carte sous forme de graphe, c'est-à-dire un réseau de nœuds. D'abord, nous avons créé la classe Node. Ainsi, chaque nœud du graphe est un objet contenant différents attributs. Le premier est sa position dans l'environnement de jeu. Il est nécessaire étant donné que le nœud n'est pas un objet physique géré par Unity, cela serait bien trop lourd à gérer pour une grande carte.

Ensuite, il possède un ensemble de voisins, allant jusqu'à 8 d'entre eux, soit un dans chaque direction, et un parent qui correspond au voisin précédent le long du chemin jusqu'au joueur.

Enfin, il comporte deux distances, celle jusqu'au départ, l'ennemi, et celle jusqu'à la destination, le joueur, qui est la distance optimale hypothétique, c'est-à-dire la distance que devrait parcourir l'ennemi s'il n'y avait aucun obstacle entre le joueur et ce nœud.

Ensuite, nous avons dû réfléchir au meilleur choix pour la création des nœuds. Nous avons finalement décidé d'instancier un nœud pour chaque unité de sol créée lors de la génération de la carte. Pour cela, nous avons créé un

script très simple rattaché à l'objet sol appelant la fonction AddNode qui instancie un nouveau nœud et le rajoute à la liste de ceux-ci. Cette solution est à la fois légère, elle ne pose donc aucun aucun problème à la création de grandes cartes, et malléable, car il suffit de retirer ce script du prototype d'un objet de sol ne permettant pas les déplacements du joueur, comme un coffre par exemple, pour qu'il ne soit pas pris en compte dans la création du graphe.

Le script AllNodes contient la liste de l'ensemble des nœuds, ainsi que la méthode AddNode évoquée plus haut. Une fois cette liste remplie pendant la création de la carte, il faut relier les nœuds voisins pour créer un graphe. Deux méthodes sont implémentées pour parvenir à cet objectif. D'abord, CreateLinksBetweenNodes applique à chaque nœud le même processus. Elle vérifie s'il existe un voisin dans chaque direction. Les quatre voisins horizontaux et verticaux, s'ils existent, sont alors ajoutés à la liste des voisins du nœud. Pour chaque voisin en diagonale, on rajoute la vérification que le nœud horizontal à la fois en contact avec le nœud actuel et le voisin existe, et de même pour celui vertical. On crée le lien seulement si c'est bien le cas. Cela permet de s'assurer que l'ennemi se déplacera en diagonale seulement s'il a la place nécessaire pour le faire.

Cette méthode appelle la deuxième, DoesNeighborExist. Celle-ci, comme son nom l'indique, vérifie si un voisin existe aux positions données en paramètre et renvoie un booléen en fonction du résultat ainsi que le nœud en question, ou la valeur "null" s'il n'existe pas.

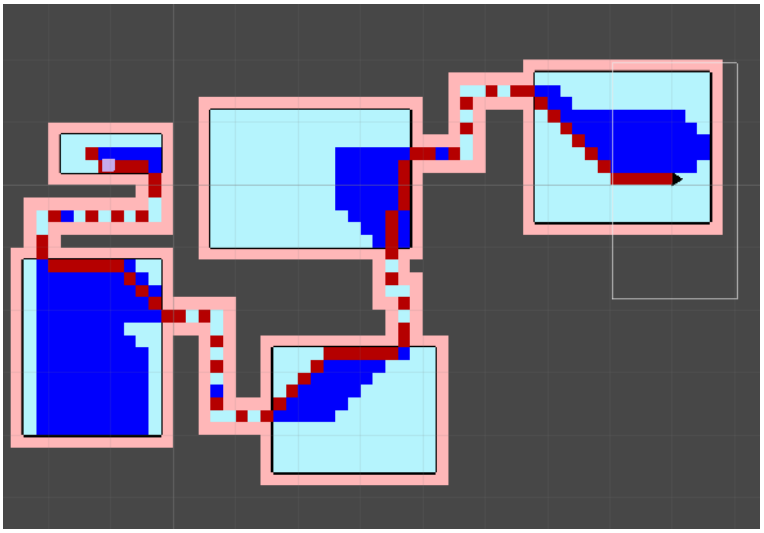
Ce script s'occupe aussi de garder à jour le nœud sur lequel se situe le joueur. Cela est nécessaire pour que les différents ennemis puissent effectuer le pathfinding jusqu'à celui-ci. Pour cela, on utilise la méthode FindNode qui prend en paramètre la position de l'objet Player, attribut du script, et recherche et retourne le nœud dont le rayon contient ces coordonnées. FindNode est appelée à chaque frame par la fonction Update. Si le nœud correspondant à la position du joueur a changé d'un appel à l'autre, l'attribut booléen PlayerPositionChanged prend la value "True" et indique que le pathfinding doit recalculer le chemin jusqu'au joueur.

Entre alors en jeu le script de pathfinding, il est un composant de chaque ennemi présent sur la carte. Sa fonction Update est, par définition, appelée à chaque frame. Cependant, elle n'effectue aucune action si l'attribut PlayerPositionChanged de AllNodes est "False". Cela permet de recalculer le chemin

que doit suivre l'ennemi seulement si le joueur a bougé et économise donc d'importantes ressources. Si le joueur s'est déplacé, le nœud correspondant à la position de l'ennemi est mis à jour à l'aide de la méthode FindNode de AllNodes, et la méthode AStar est appelée. Celle-ci, dont nous avons parlé précédemment, calcule le plus court entre l'ennemi et le joueur puis met à jour l'attribut Path, qui garde en mémoire la liste des nœuds à suivre pour atteindre le but. Pour cela, elle calcule, pour chaque voisin du nœud de départ, la potentielle distance jusqu'au nœud d'arrivée et modifie son attribut Parent. Ainsi, elle fait la somme de la distance depuis le départ et l'hypothétique distance optimale jusqu'à la destination, c'est-à-dire la distance sans obstacle, appelée heuristique. Une fois cela fait, elle répète la même action sur le nœud dont la distance calculée est la plus faible, soit celui qui amènera probablement l'ennemi au joueur le plus rapidement. À chaque fois que la distance d'un même nœud est recalculée, son attribut Parent est modifié si un nouveau nœud lui offre une plus faible distance jusqu'au déppart. Finalement, quand la destination est atteinte, la méthode ReconstructPath est appelée. Elle met à jour la liste de nœud constituant le plus court chemin en partant de la destination et en remontant tous les parents jusqu'au nœud d'origine. L'ennemi est alors prêt à continuer sa route jusqu'au joueur.

Enfin, l'ennemi connaît le chemin à suivre pour atteindre le joueur. Il doit maintenant se déplacer le long de cette voie. Pour effectuer cette action, nous avons créé le script EnemiesMovement. Celui-ci consiste en une fonction permettant d'appliquer à l'ennemi une translation dans la direction du prochain nœud du chemin, à chaque frame et en fonction de sa vitesse, contenue dans l'attribut Speed. Lorsqu'il atteint le nœud, celui-ci est supprimé de la liste et l'ennemi se déplace vers le suivant. Pour finir, l'ennemi arrête de bouger lorsqu'il se situe à portée d'attaque du joueur, valeur de l'attribut Range.

Finalement, l'algorithme de pathfinding en lui-même s'est avéré être la partie la plus simple. Les réelles difficultés résidaient d'abord, dans le choix de l'implémentation permettant de représenter la carte sous forme de graphe. En effet, de nombreuses possibilités s'offraient à nous et il ne fut pas simple d'identifier la plus optimale et adaptée à notre projet. Aussi, faire bouger l'ennemi le long du chemin calculé par l'algorithme A* fut plus compliqué qu'il ne pourrait paraître du fait de notre manque de connaissances concernant Unity au premier abord.

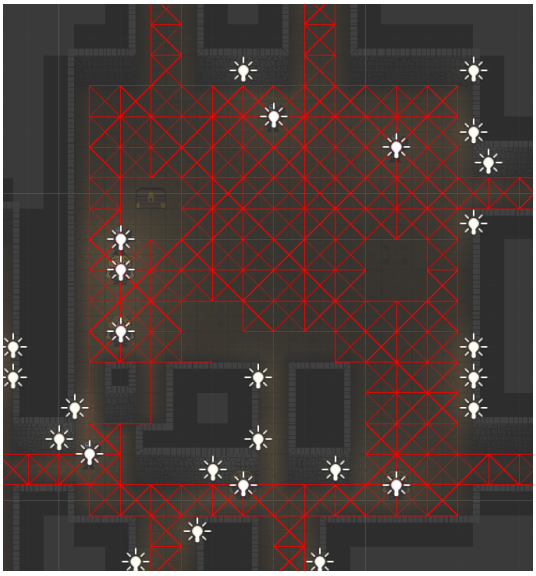


*Calcul d'un chemin grâce au pathfinding A**

Dans un deuxième temps, nous devions donc apporter diverses améliorations à notre IA jusqu'alors primaire. D'abord, il fallait implémenter la possibilité de créer des ennemis "à la volée". Ensuite, il fallait gérer la génération de ceux-ci en fonction de la carte. Enfin, il fallait modifier le pathfinding afin que les ennemis puissent gérer plusieurs joueurs et ainsi accueillir la fonctionnalité de jeu en multijoueur.

Tout d'abord, avant de commencer à remplir nos objectifs pour cette soutenance, nous avons grandement optimisé l'implémentation de la génération du graphe nécessaire à l'algorithme de pathfinding. En effet, l'implémentation précédente, bien que fonctionnelle, était très gourmande en ressources. Il était relativement long de calculer le graphe pour une carte de taille conséquente. Ainsi, en profitant de la grille de booléens offerte par l'implémentation de la génération de la carte, nous avons pu calculer le graphe de manière optimisée. Nous avons créé une grille de la taille de la carte contenant les noeuds du graphe. Les positions des noeuds correspondent aux positions sur lesquelles les ennemis peuvent se déplacer en prenant à la fois en compte la perspective et les nouveaux obstacles amenés avec l'amélioration de la génération de la carte. En même temps que les noeuds sont créés, les liens le sont aussi, afin d'éviter un parcourt supplémentaire de la grille. Ainsi, à chaque nouvelle ligne, les liens sont créés vers les noeuds de la ligne précédente et vers le précédent noeud de la même ligne. Cela se trouve être bien plus optimal que de vérifier les 8 voisins potentiels en parcourant les noeuds. Enfin, la génération de la carte faisant qu'il ne peut pas y avoir de sol aux extrémités de la grille,

de nombreux tests sont économisés. Il est maintenant possible de générer de très grandes cartes sans aucun problème.



Représentation du graphe

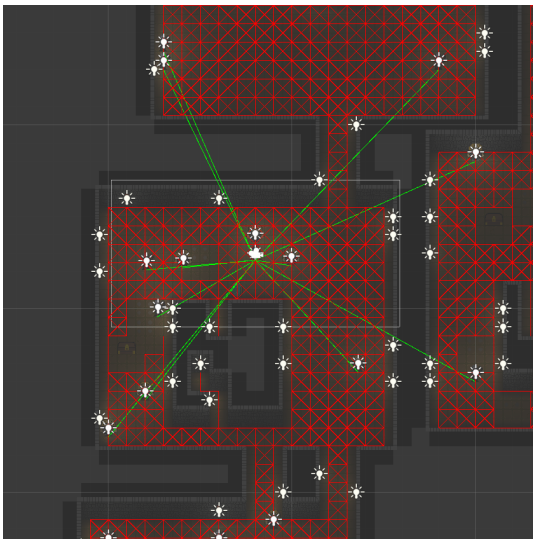
Suite à cela, il a fallu créer une réelle implémentation des ennemis qui soit suffisamment généralisée pour être malléable. Le but de celle-ci est donc de permettre la création de différents ennemis de manière assez simple, au niveau algorithmique tout du moins. Pour accomplir cet objectif, les Objets Scriptables, déjà évoqués plus haut, se sont avérés être encore une fois un outil parfaitement approprié. Nous avons créé une Prefab générique contenant les différents scripts et composants nécessaires au bon fonctionnement des ennemis. Celle-ci ne contient pas les informations de l'ennemi comme sa texture ou ses différentes statistiques. Toutes ces informations sont contenues dans l'objet scriptable de chaque ennemi. Ce dernier fait donc le lien entre les scripts, chacun d'eux manipule directement les variables contenues dans celui-ci. Cela permet à chaque script d'accéder aux variables à jour avec les autres scripts. Ainsi, pour générer des ennemis, il suffit de créer une nouvelle instance de la Prefab Enemy, de créer une nouvelle instance de l'objet scriptable de l'ennemi souhaité - afin que tous les ennemis du même type ne partagent pas les mêmes variables de vie - et de récupérer les composants de type Script pour leur assigner l'objet scriptable nouvellement créé. Enfin, pour créer un nouveau type d'ennemi, il suffit de créer un objet scriptable et de renseigner ses statistiques.

Une fois l'implémentation des ennemis finie et la possibilité de les générer, il est temps de le faire. Pour cela, nous récupérons la grille de booléens et la liste des salles de la génération de la carte. Afin d'être cohérent avec le mode multijoueur, les ennemis sont générés en même temps que la carte à chaque étage. Ainsi, tous les joueurs connectés en réseau ont la même carte, mais aussi les mêmes ennemis aux mêmes positions, car la partie est générée à partir de la même graine. Cela assure que le comportement des ennemis soit le même pour tous les clients connectés, car les actions des ennemis sont calculées par ces derniers en local afin de ne pas communiquer de nombreuses informations inutiles à travers le serveur. Actuellement, dans chaque salle, deux ennemis de type aléatoire sont créés à des positions aléatoires, en s'assurant qu'un autre ennemi n'existe pas déjà à ladite position.

Finalement, lors de la première soutenance les ennemis n'avaient besoin de gérer qu'un seul joueur et suivait celui-ci à travers toute la carte afin de bien mettre en avant le fonctionnement de l'algorithme de calcul de chemin. Pour cette soutenance ce n'est plus le cas. Les ennemis doivent maintenant être capable de prendre pour cible l'un des joueurs pendant les parties en réseau. De plus, ils doivent rester aux alentours de leur salle de création. En effet, tout d'abord, le fait que l'ensemble des ennemis de l'étage se dirigent tous vers le joueur n'aurait aucun sens. Ensuite, ce serait rapidement insoutenable du point de vue des ressources nécessaires au calcul de tous les chemins pour de grandes cartes contenant de nombreux ennemis. Nous avons donc totalement modifié le comportement de l'ennemi.

Maintenant, l'ennemi ne se déplace qu'à partir du moment où il a été "déclenché", c'est-à-dire lorsque le joueur est entré dans la zone de détection de l'ennemi tout en n'étant pas séparé de celui-ci par un obstacle. Une fois qu'un ennemi a été déclenché, il se déplace vers le joueur à chaque fois que celui-ci pénètre la zone de détection. Celle-ci consiste un cercle de centre la position de génération de l'ennemi et de rayon variable selon le type d'ennemi. Pour un déplacement plus fluide, les ennemis se déplacent de deux manières différentes. Lorsque le joueur n'est pas en vision directe de l'ennemi, c'est-à-dire lorsqu'un obstacle les sépare, ce dernier suit le chemin calculé par l'algorithme de pathfinding et ceux jusqu'à avoir le joueur dans son champs de vision. Ensuite, l'ennemi se déplace de manière linéaire vers le joueur, sans se soucier des noeuds et du chemin calculé, jusqu'à être à portée d'attaque du joueur, donnée variable d'un ennemi à l'autre. Cela permet un déplacement à la fois plus logique, le plus court chemin dans cette situation étant la ligne droite, et moins "robotique", ou plus naturel, car l'ennemi ne suit pas une

forme géométrique formée par la liaison de plusieurs noeuds. Enfin, si le joueur sort de la zone de détection, l'ennemi utilise le même procédé pour retourner à la position à laquelle il a été généré.



Multiples ennemis détectant le joueur

Enfin, l'ennemi gère maintenant plusieurs joueurs. Pour cela, l'ennemi calcule quels sont les joueurs dans sa zone de détection parmi l'ensemble des joueurs. Puis, parmi ceux-ci, si certains d'entre-eux sont dans son champs de vision, alors il se dirige de manière logique vers le plus proche. Sinon, il se dirige vers le premier joueur se trouvant dans sa zone de détection et vers lequel il peut calculer un chemin. Il n'est pas nécessaire dans ce cas là de calculer lequel des joueurs peut être atteint en suivant le plus court chemin car à partir du moment où l'un des joueurs sera en vision directe, il se dirigera vers celui-ci.

Dans un dernier temps, nous avons ajouté les dernières améliorations afin d'atteindre nos objectifs ajustés suite à la deuxième soutenance.

D'abord, nous avons amélioré la génération des ennemis. Celle-ci prend maintenant en compte la taille de la salle lors du choix du type d'ennemi. Ainsi, les ennemis à distance seront privilégiés dans les salles les plus grandes et ceux plus résistants dans les salles plus petites.

Ensuite, nous avons implémenté l'amélioration des statistiques en fonction du niveau du joueur et de l'étage dans lequel il se trouve. L'ensemble des ennemis de l'étage étant généré lors de la création de celui-ci, pour permettre le

bon fonctionnement du multijoueur, nous ne pouvions pas calculer les statistiques des ennemis à ce moment là. En effet, le joueur peut gagner plusieurs niveaux en un seul étage et nous souhaitons que les ennemis suivent ce niveau. Cependant, il n'était pas non plus possible d'augmenter les statistiques de tous les ennemis de l'étage à chaque fois que le joueur passe un niveau. Nous risquions une perte de fluidité du jeu à chaque passage de niveau pour de grands étages contenant beaucoup d'ennemis et nécessitant donc beaucoup de calculs pour ajuster toutes les statistiques. Nous avons finalement décidé de calculer les statistiques de chaque ennemi lors de sa première activation, c'est-à-dire la première fois que le joueur entre dans sa zone de détection. Ainsi, les statistiques de chaque ennemi ne sont calculées qu'une seule fois et cela reste cohérent par rapport au niveau du joueur car le décalage entre celui-ci et celui de l'ennemi ne peut être que minime.

Enfin, nous avons enrichi notre jeu de quelques ennemis pour offrir un peu de diversité au joueur. Pour cela, nous avons profité de l'implémentation des ennemis sous forme d'objets scriptables, que nous avions précédemment mis en place, qui a réellement simplifié cette tâche.

6 Expérience individuelle

6.1 Thomas

Ce projet fut, dans tous ses aspects, une expérience très enrichissante. Même si nous n'avons pas pu réaliser tous nos objectif je reste très content du résultat de nos efforts. J'ai pu expérimenter avec des technologies intéressantes dans le réseau et le site web étant donné la carte blanche que nous avions de ce côté là, et ait beaucoup appris. Au cours de ces derniers moi je me suis posé beaucoup de questions, mais n'ait jamais eu la sensation de perdre le fil conducteur de ce projet. En tant que chef de projet, j'ai aussi acquis de l'expérience dans la gestion d'un groupe et la prioritisation de tâches pour obtenir des résultats concrets. Avec le recul, si je devais refaire ce projet il serait un peu différent, ferait des choix un peu différents, mais comme dit précédemment, pour une première fois je trouve le résultat très bon. Ce projet en groupe m'a aussi beaucoup appris à compter sur mes camarades, j'avais plutôt l'habitude d'être seul/faire une grande partie du travail lors de mes projets de programmation, mais dans ce projet les autres membres ont totalement dépassé mes attentes, chacun rivalisant d'ingéniosité. Nous partagions nos connaissances aux autres membres du groupes, comme par exemple moi pour Git, ou encore William et l'architecture des "ScriptableObject" qui a permit d'obtenir un projet Unity très propre dans son ensemble. C'était une ambiance formidable, et je suis très content d'avoir pu réunir et faire ce projet avec cette équipe. Chacun était indispensable à sa façon, notre complémentarité n'en étant que renforcée.

"Il suffit d'ingurgiter entre amis une poêlée de petites pommes de terre sautées pour venir à bout de tous les défis" – Platon

6.2 Adrien

Nous voilà, déjà, enfin, c'est relatif, arrivés à la fin de ce projet. Ce fut une expérience très enrichissante. Pour mener à bien la réalisation de ce jeu, je suis passé par la recherche, l'hésitation, la prise de décision, la création, mais aussi la destruction, trop souvent l'échec mais finalement la réussite... Autrement dit, j'ai découvert toutes les étapes nécessaires à la réalisation d'un projet de bout en bout. D'une part, cela m'a permis d'emmagasiner de nombreuses connaissances concernant la programmation. Mon expérience dans ce domaine se limitant à la création d'un Snake en Python et aux travaux pratiques des ACDCs, mes lacunes n'avaient d'égal que nos nombreux

souhaits de mécaniques à implémenter. Mais c'était loin d'être insurmontable, il a simplement fallu passer par une importante phase d'apprentissage. D'autre part, cela m'a appris à gérer un projet en groupe. J'ai appris à gérer mon temps par rapport à nos prévisions et aux dates limites, mais aussi mes priorités, que ce soit au sein du projet entre différentes tâches à accomplir ou entre le projet et les révisions des autres matières. Enfin, je suis heureux du résultat que nous avons atteint et de tout ce que ce projet m'a apporté.

> *"La théorie, c'est quand on sait tout et que rien ne fonctionne. La pratique, c'est quand tout fonctionne et que personne ne sait pourquoi. Ici, nous avons réuni théorie et pratique : Rien ne fonctionne... et personne ne sait pourquoi !"*– Albert Einstein

6.3 Eliès

Ainsi s'achève ce premier projet de groupe de notre scolarité. Je pense que, pour mes camarades comme moi, ce projet nous a permis de nous confronter à une première approche du métier d'Ingénieur. Gestion du temps, planning d'avancement, réunions ou présentation de notre projet, la philosophie de notre future carrière semble déjà avoir une certaine esquisse au travers de notre jeu. Sur le plan personnel, je pense que ce projet m'aura servi à développer mes compétences scientifiques mais également mon rapport avec les autres. En effet, mis à part les TPE au lycée, l'expérience de travail de groupe ne s'est que très rarement présentée au cours de mon cursus scolaire. Enrichissant et bénéfique, ce projet m'a permis de découvrir la vie en groupe, la confiance entre chacun, l'entraide et l'envie de se dépasser pour atteindre les objectifs fixés. Tous ces attraits qui font, au final, le métier d'Ingénieur, un véritable croisement entre gestion scientifique et humaine.

> *"Do or not do. There is no try."*– Master Yoda

6.4 William

Ce projet fut un travail conséquent, nous avions mis la barre très haute, voire même un peu trop puisque nous avons dû revenir sur certaines de nos décisions, cependant cela fait partie d'un projet et ce fut une bonne expérience que de devoir revenir sur nos prévisions. De plus, ce fut une très bonne expérience, que ce soit sur la répartition des tâches la manière d'appréhender les dead-lines et plus généralement tout ce qui tourne autour d'un long travail

d'équipe. Personnellement, ce projet m'a permis de m'améliorer en programmation, bien que nous le faisions en cours, cette fois-ci était différente puisque j'étais complètement libre de faire ce que je voulais, cela m'a été très bénéfiques. De plus, j'ai pu apprendre et découvrir de nombreux algorithmes très intéressant qui me serviront très certainement plus tard. Plus généralement ce projet fut une expérience importante dans de nombreux points, que ce soit sur la programmation, l'algorithme ou encore le travail de groupe. Cette expérience me sera très utile pour les futurs projets que je devrais mener, qu'il soit personnel ou en groupe.

"I am not a goddamn Turtle, or am I ?"– TurtleSmoke

7 Conclusion

Bien que tous les points de notre cahier des charges n'aient pas été respectés, le projet final reste grandement à la hauteur de nos attentes. Après trois soutenances de travail intensif, nous avons réussi à créer un jeu fonctionnel et dynamique. Ce premier travail de groupe nous a tous apporté, autant en matière de connaissances que de gestion de projet et conclut ainsi notre première année à Epita mais marque aussi le début de nos études en tant qu'Ingénieur dans le Numérique.