Once Upon a Time

Alexandre LEDOUX
Lucas MOREL
Mantas KRUKONIS
Nicolas SIMEONOV
Anir PERROD

Sommaire

1 Présentation du projet

1.1 Introduction

Avec une réelle volonté de transmettre la passion des contes des frères Grimm au plus grand nombre, le studio The Grimms a eu l'idée de se lancer dans le développement d'un jeu vidéo, pour permettre à tous de découvrir ou redécouvrir les fabuleux contes des frères Grimm. Cette idée est née d'un constat simple : de plus en plus de jeunes passent leurs loisirs à jouer aux jeux vidéo, et cela au détriment de la lecture. C'est ainsi que nous est venue cette idée : développer notre jeu vidéo dans lequel chaque joueur incarne le personnage principal du conte afin de mêler le plaisir de jouer et la culture des contes.

1.2 Présentation de notre jeu

Pour garder le joueur sur notre jeu, il nous est impératif de trouver de multiples moyens d'éveiller sa curiosité et son envie de continuer à jouer. Pour ce faire, nous avons décidé de représenter chaque conte par un niveau indépendant. Cela permettra au joueur d'être plongé dans un univers unique, en lien avec le conte concerné.

À terme, notre jeu sera composé de 2-3 niveaux, avec chacun un objectif différent, écrit en fonction du conte qui lui est associé. Nous avons la volonté de créer des actions et méthodes de jeu différentes et exclusives sur chaque conte. Par exemple, dans le premier niveau, représentant le conte du Petit Chaperon rouge, nous voulons pouvoir prendre une épée dans un coffre pour combattre le loup. Dans un autre conte, on pourrait résoudre des énigmes, faire des choix ou chercher des objets cachés.

1.3 Fonctionnement d'une partie

Lorsqu'un joueur arrive sur notre jeu, nous voulons qu'il puisse créer ou rejoindre une partie avec un autre joueur en ligne. S'il crée la partie, alors il fait le choix du niveau sur lequel il veut jouer. Si le joueur décide de rejoindre une partie, il récupérera toutes les informations de la partie en cours (coordonnées des autres joueurs, emplacement des coffres, points de vie de chacun, ...).

Durant la partie, l'ensemble des membres doit s'allier pour trouver des objets, des chemins et autres avantages qui leur permettront de terminer le niveau. La partie se terminera si l'ensemble des joueurs réussit à atteindre l'objectif final du niveau concerné, tel qu'accéder à la maison de la grand-mère pour Le Petit Chaperon rouge.

1.4 Planning global

En début d'année, lors de la première soutenance, nous avions présenté un planning global permettant de présenter nos attentes vis-à-vis de la livraison des différents modules du jeu. Ce planning est découpé en 9 principaux organes :

- Le site web : Permet de présenter notre projet, notre équipe et la possibilité de télécharger le jeu, tout cela accessible sur internet à l'adresse suivante : the-grimms.com.
- Le gameplay : Représente l'ensemble des actions réalisables par le joueur. Il est la partie visible par les joueurs. Le gameplay nécessite d'être soigné pour donner envie de jouer.
- Le multijoueur : Le développement d'un système permettant de synchroniser l'ensemble des données entre tous les joueurs d'une même partie.
- L'intelligence artificielle : Il s'agit de l'ensemble des actions automatisées, dont aucun joueur n'est à l'origine et dont les calculs sont faits sur l'ordinateur d'un client choisi en fonction de certaines conditions.
- Les menus : Il s'agit de la page sur laquelle arrive le joueur lorsqu'il lance le jeu. À terme, il sera possible de créer une partie sur un niveau précis ou bien de rejoindre une partie en cours.
- Les graphismes : Ce sont l'ensemble des éléments visuels qui permettent de créer une ambiance particulière vis-à-vis des contes. Nous avons fait le choix de graphismes légèrement pixélisés pour notre jeu.
- L'interface : Cela regroupe l'ensemble des boutons, textes et inventaires présents sur la fenêtre lorsque l'on joue une partie.
- Les sons et musiques : Pour ajouter du dynamisme durant la partie, nous souhaitons à terme ajouter des effets sonores pour plus de détails.
- Les tests : Cette partie est effectuée à la fin du développement du jeu, afin de s'assurer que tout fonctionne correctement et d'ajuster si nécessaire quelques détails.

L'ensemble de ces sous-parties forme l'ensemble complet de notre jeu. En effet, en associant l'intégralité de ces développements, nous obtiendrons notre résultat final.

Afin de maintenir nos objectifs vis-à-vis des dates de livraison, il nous est nécessaire de travailler régulièrement sur le développement. Il est également indispensable de se répartir correctement le travail, comme nous l'avons fait (voir cahier des charges technique).

Certaines tâches nécessitent plus de temps que d'autres, ce qui explique la différence de temps que nous avons prévue entre chacune d'elles. En effet, le développement du multijoueur et du gameplay sont les piliers de notre jeu. Sans

eux, nous n'aurions pas de projet à proposer. C'est pour cela que nous avons fait le choix de consacrer la majorité de notre temps de développement à ces deux sujets.

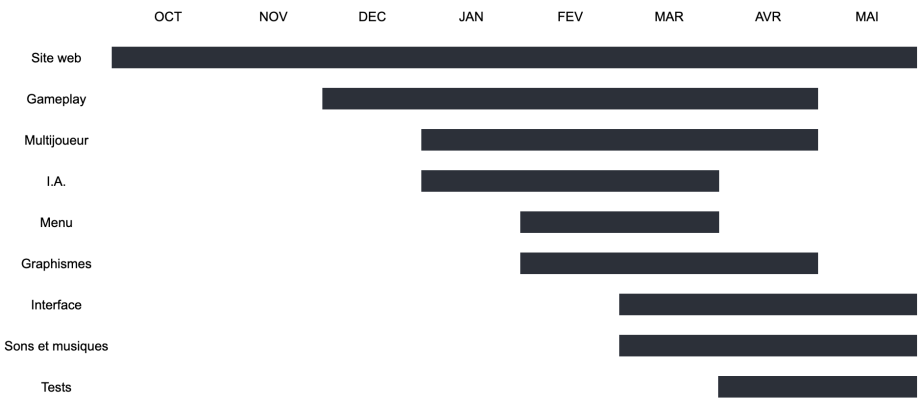

Figure 1: Planning global

À l'heure actuelle, l'ensemble des dates de développement des 9 principaux domaines est respecté, mis à part pour l'intelligence artificielle, qui ne répond toujours pas à nos attentes et sur laquelle nous allons consacrer davantage de temps. En effet, avec la volonté d'avoir une architecture stable et solide, nous avons pris davantage de temps pour la réalisation du multijoueur, ce qui nous a empêchés d'avancer comme nous le souhaitions sur la partie de l'intelligence artificielle.

2 Ce que nous avons réalisé

2.1 Système de multijoueur

2.1.1 Choix de la technologie

Lors des débuts du développement de notre jeu, nous avions créé un objet 2D représentant notre joueur. Nous l'avions relié à un script qui nous permettait de gérer ses déplacements. Après réflexion et prise en compte de la nécessité d'intégrer du multijoueur dans notre jeu, nous avons dû repenser l'intégralité de l'architecture de notre projet. En effet, nous devions penser à la possibilité de représenter un nombre variable de joueurs dans notre jeu. Pour cela, nous avons pensé à transformer notre joueur 2D en un préfabriqué que l'on clonerait dans le code pour représenter chaque joueur.

Pour réaliser notre système de multijoueur, nous avons réfléchi entre deux méthodes différentes. Notre première idée était d'utiliser une librairie intégrée directement à Unity, permettant d'envoyer des messages en temps réel entre les différents joueurs d'un même réseau local. Cependant, cette méthode comportait de multiples contraintes. Tout d'abord, si deux joueurs voulaient jouer entre eux, il aurait été nécessaire qu'ils soient connectés sur le même réseau local. Cela empêchait donc de jouer avec quelqu'un d'autre sur internet. De plus, étant une méthode client/serveur, si deux joueurs sur le même réseau décidaient de jouer ensemble, le premier joueur serait obligé d'héberger la partie sur son ordinateur, et les autres joueurs se connecteraient à sa machine. Pour envoyer des informations entre chaque joueur, il aurait été nécessaire que le client envoie un message au client/serveur, qui, lui, s'occuperait de le renvoyer à tous les autres clients. Il s'agit donc d'un système centralisé dont l'un des clients est responsable. Un problème se pose alors : si le joueur hébergeant la partie quitte le jeu, tous les autres joueurs sont déconnectés et plus personne ne peut continuer à jouer dans la partie sur laquelle ils se sont tous rassemblés, et ont sans doute pris du temps à évoluer dans le niveau.

Après avoir réfléchi sur le sujet, nous ne voulions pas avoir un jeu uniquement jouable entre différents joueurs d'un même réseau local. Nous avons donc décidé de créer notre propre système de multijoueur.

Pour ce faire, nous avons décidé de développer une API qui s'occupe de recevoir des requêtes HTTP et de faire lui-même une requête à un serveur Node.js qui s'occupe d'envoyer ce message à tous les clients connectés de la partie.

2.1.2 Présentation de l'API

Notre API comporte 2 tables principales et une table mineure. La première, nommée Room, s'occupe d'enregistrer l'existence d'une partie en ligne. La seconde, nommée Participation, représente la participation d'un client dans cette partie. Une Room possède donc autant de Participations que souhaité, et Participation est reliée à une seule Room. La dernière table est Device, elle permet d'enregistrer un appareil et de lui attribuer un token.

Room représentant la partie, il est nécessaire d'enregistrer des informations sur celle-ci. Pour ce faire, elle possède trois champs. Le premier est un Id unique. Le second est un token unique permettant de le partager aux autres joueurs afin qu'ils puissent rejoindre la partie. Le dernier champ est data, de type JSON. Ce champ enregistre toutes les informations concernant une partie. Nous avons fait le choix de mettre un type JSON car toutes les parties ne possèdent pas les mêmes informations. Par exemple, dans le niveau du Petit Chaperon rouge, nous voulons qu'il y ait un coffre avec dedans une épée et une tarte, et plus loin, un loup avec 200 points de vie et une maison. Dans le second conte, Hansel et Gretel, nous ne voulons pas de coffre, mais d'autres objets.

La seconde table est Participation et possède trois champs. Le premier est un Id unique permettant de le différencier des autres, une relation avec une Room et un champ data de type JSON qui permet d'enregistrer les informations concernant ce joueur. Par exemple, data enregistre les coordonnées X et Y, s'il a un objet dans la main droite, si oui, lequel avec ses attributs et plein d'autres informations qui peuvent être ajoutées en fonction du conte.

Device est la dernière table, elle possède deux champs : le premier est un Id unique et le second un token de connexion unique, permettant d'effectuer l'authentification du client qui fait une requête.

2.1.3 Présentation de Pusher

Pour faire notre système de synchronisation en temps réel, le client nous envoie les nouvelles données, mais nous devons ensuite les répandre à tous les clients de la partie. Cependant, donner les nouvelles informations à celui qui est à l'origine de la requête est possible, mais communiquer des informations à d'autres clients n'est pas possible, à moins qu'ils ne fassent des requêtes à intervalles de temps réguliers sur le serveur pour voir si quelque chose a changé. Cette dernière méthode étant trop gourmande en termes de ressources, nous avons fait le choix d'utiliser un système de WebSocket. Un WebSocket fonctionne par le biais d'un serveur Node.js, qui peut envoyer des informations du serveur aux clients connectés à un channel auquel ils se sont abonnés. Ne voulant pas développer notre serveur Node.js, nous avons fait le choix d'utiliser les services de Pusher. Il s'agit d'une plateforme sur laquelle on peut faire des requêtes via leur API avec trois informations spécifiques : le channel concerné, l'événement concerné et les données à envoyer. La plateforme s'occupe de recevoir notre requête et de l'envoyer à tous les clients abonnés à ce channel, dans notre cas, nos joueurs en ligne. Cette plateforme nous propose un plan gratuit, limitant le nombre de messages à 200 000 par jour. Cela est largement suffisant pour notre jeu.

L'ensemble de la structure de notre système de multijoueur est présenté ci-dessous. On peut y voir notre API et celle de Pusher, ainsi que trois clients, représentant chacun un joueur qui jouerait au jeu dans une même partie.

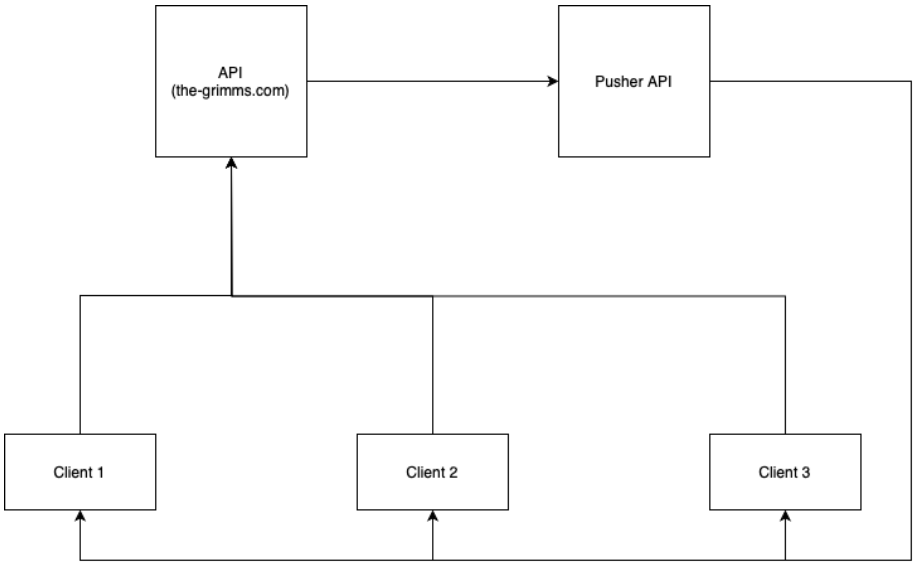

Figure 2: Organisation de l'API

2.1.4 Présentation des points d'entrées de notre API

Afin d'utiliser les services de notre API, différents points d'entrée sont mis à disposition des clients. Il existe différentes méthodes : les requêtes POST, qui permettent de créer un nouvel élément en base de données, et les requêtes GET, qui permettent de récupérer des informations en base de données. Actuellement, nous avons limité ces deux méthodes, car nous ne voyons pas l'utilité d'ajouter des DELETE ou encore des PUT.

Lorsque le joueur lance le jeu pour la première fois, il fait une requête en POST sur l'API pour demander un token d'authentification. Le serveur le crée en base de données et lui retourne le token, que le client enregistre dans la mémoire d'Unity pour l'utiliser lors de ses prochaines requêtes.

Quand un joueur décide de créer une partie, il effectue une requête en POST sur l'API avec des données spécifiques qui vous seront présentées dans la sous-partie suivante. Cela enregistre les données et retourne le token de la partie, ce qui permettra au joueur de le partager avec ses amis pour qu'ils puissent rejoindre cette partie.

Après avoir créé une partie ou s'il veut rejoindre une partie, dans ces deux cas, il est nécessaire de créer une participation en base de données. Pour ce faire, le client envoie une requête en POST avec des attributs. Il est nécessaire de remplir data avec des données en JSON (x, y, objet dans la main, ...). Pour obtenir ce JSON, le client va générer son personnage en C# et va le sérialiser en JSON pour l'envoyer au serveur. Toute cette partie de transformation des objets C# en JSON vous sera présentée ultérieurement.

Le point d'entrée principal de l'API est " /messages ". Il s'agit d'un point

d'entrée acceptant les requêtes en POST et qui prend en paramètre le type de l'événement à traiter : nouveau participant " event-newParticipation ", mise à jour des informations d'un participant " event-participation ", mise à jour des informations de la Room " event-room " ou la déconnexion d'un joueur " event-disconnect ". Il prend également l'Id de la Room ou de la Participation concernée et data, le JSON des nouvelles données à remplacer. L'API traite en fonction de l'événement. S'il s'agit de l'ajout d'un nouveau participant, nous envoyons le nouveau participant avec l'événement associé et les données de notre participant à Pusher. S'il s'agit de la modification d'une participation ou de la Room, nous modifions en base de données le champ data correspondant et envoyons à Pusher les nouvelles informations sur l'événement dédié. Si l'événement est de type déconnexion, nous supprimons la participation concernée de la base de données, et également la Room s'il s'agissait du dernier joueur, puis envoyons à Pusher une nouvelle requête.

2.1.5 Transformation des objets C# en JSON et inversement (Serialization, Désérialization)

Dans notre code C#, tous sont représentés sous la forme d'objets. Cela permet d'avoir des attributs communs, des méthodes communes et un code propre et optimisé. Cependant, lorsque nous voulons envoyer toutes les informations d'un objet vers notre API, il nous est nécessaire de le transformer en JSON. C'est ainsi qu'entre en jeu la sérialisation. Une méthode C# nous permet de le faire très simplement. Cependant, certains attributs ne sont pas sérialisables. Par exemple, dans notre objet Participation, qui représente un joueur dans la partie, nous avons un attribut GameObject de Unity qui permet de l'afficher dans le jeu. Cet attribut comportant de très nombreuses informations et étant complexe, il n'est pas possible de le sérialiser. Pour l'objet Participation, nous n'envoyons donc pas de GameObject, mais ce n'est pas grave, car les autres clients pourront l'instancier de leur côté avec les autres informations qui composent cette participation (x, y, image de représentation).

Lorsqu'un événement est appelé par le serveur, par exemple pour la mise à jour des informations d'une Participation, nous recevons du JSON et devons le faire correspondre avec le joueur concerné. Nous désérialisons donc ce JSON, instancions un nouveau GameObject pour la Participation et mettons cet objet là où il était auparavant.

Ce travail de sérialisation/désérialisation est capital dans le bon fonctionnement de notre jeu. En effet, avec des méthodes simples et rapides, il nous est facile de mettre à jour des informations, en appelant une simple fonction, ce qui nous permet d'avancer rapidement dans le développement du gameplay.

2.1.6 Présentation de la création d'une Partie

Lorsqu'un joueur veut commencer une nouvelle partie, il clique sur le bouton Nouvelle partie et choisit son niveau. S'il choisit le niveau 1, correspondant au Petit Chaperon rouge, cela crée un objet Level1 en C#, avec des infos comme

une liste d'objets (coffre, maison, ...) et des informations par défaut comme les coordonnées x et y qui seront attribuées de base à tous les nouveaux participants. Lorsque le client a cet objet Level1, héritant de l'objet Game, il sérialise toutes ces infos en JSON et les met dans la requête POST pour la création de la Room. Toutes ces informations communes à tous les joueurs seront ainsi synchronisées sur le serveur.

2.1.7 Présentation pour rejoindre une Partie

Après avoir créé la partie ou s'il veut rejoindre une partie existante, le joueur doit créer sa participation sur le serveur. Pour ce faire, il prend le token de la Room associée et fait une requête sur le point d'entrée POST pour la création d'une Participation avec le token. L'API informera tous les participants de ce nouvel arrivant et répondra dans sa réponse HTTP toutes les informations de la partie à charger sur le jeu de l'arrivant. Ainsi, tous les joueurs sont synchronisés et ont les mêmes informations à l'écran.

2.1.8 Avantage de notre système de multijoueur

Avec notre API et notre système de multijoueur, nous avons de nombreux avantages que nous n'aurions pas eus avec une librairie fermée. Tout d'abord, nous pouvons centraliser toutes les données d'une partie en cours dans une base de données, accessible sur internet, ce qui permet de continuer une partie même si celui qui l'a créée quitte la partie alors qu'il y a d'autres joueurs dessus. Nous pouvons centraliser toutes ces informations de manière bien organisée.

2.1.9 Inconvénients de notre système de multijoueur

Le premier inconvénient de notre système est que nous sommes obligés d'être connectés à internet pour pouvoir jouer. Nous n'avons pour l'instant pas développé de mode local qui permettrait de jouer tout seul sans connexion internet. Le second problème est le nombre de requêtes effectuées. En effet, dès qu'un joueur se déplace, cela envoie une requête à l'API pour lui donner ses nouvelles coordonnées, tout comme lors de la modification de la partie (ouvrir un coffre, prendre l'épée du coffre, ...). Tout cela est gourmand en bande passante car il y a souvent des changements d'état dans le jeu.

2.1.10 Solutions pour optimiser le système de multijoueur

Pour faire face à ces nombreuses requêtes, par exemple à chaque déplacement d'un joueur, nous avons choisi d'envoyer des messages, dont le nombre est restreint ou non. Quand un joueur se déplace, cela fait appel à la fonction pour effectuer une nouvelle requête à l'API, mais comme cette catégorie est restreinte, cela n'envoie qu'une requête sur 150. Quand on change les informations de la carte, ce qui n'arrive pas souvent, on ne la restreint pas, ce qui permet de l'envoyer dans tous les cas. Cette méthode permet d'économiser 150 requêtes à chaque requête de déplacement. Cependant, moins il y a d'informations sur

le déplacement, moins les joueurs se déplaceront de manière fluide. Pour pallier cela, nous avons trouvé la solution de créer un mouvement fluide entre sa dernière position et sa nouvelle position. Ainsi, ses déplacements sont mis à jour tous les 150 frames, ce qui est assez raisonnable. De plus, nous avons trouvé une solution pour rendre le déplacement d'un point A à un point B de manière fluide, ce qui rend les problèmes de synchronisation invisibles à l'œil nu. Cette optimisation nous permet d'éviter de consommer trop rapidement nos 200.000 messages Pusher, mais aussi d'éviter de surcharger le serveur de l'API. Une autre optimisation intéressante pour l'avenir serait de créer notre API avec un serveur Node.js, nous permettant directement de gérer à la fois les requêtes reçues par les différents clients et d'émettre les messages WebSocket directement par lui-même. Cela nous contraindrait donc à développer à nouveau notre API à partir de zéro, mais cela nous ferait gagner le temps que notre API prend pour contacter le serveur de Pusher. De plus, nous n'aurions plus de limite sur le nombre de messages, nous serions juste limités par la bande passante descendante des clients. Cette option étant longue à mettre en place, elle n'est pas le sujet principal de nos développements futurs.

2.1.11 Sécurisation de l'API

Notre API étant le pilier de la synchronisation et de l'enregistrement de nos données, il est nécessaire de la sécuriser. Étant ouverte sur internet, il nous est important de trouver des moyens d'éviter les requêtes parasites générées par des scripts ou des bots qui pourraient créer des parties à l'infini. Pour ce faire, nous avons intégré une clé secrète dans le code source C# qui est envoyée dans toutes les en-têtes des requêtes vers l'API. Ainsi, si le serveur ne possède pas cette en-tête, cela signifie que celui qui émet la requête ne provient pas d'un joueur du jeu. Nous pouvons donc affirmer avec certitude qu'il s'agit d'un bot malveillant ou d'un autre script ayant pour objectif de surcharger le serveur. Dans ce cas, le serveur de l'API bloque l'adresse IP de l'initiateur de la requête, afin qu'il ne puisse plus continuer. Cette sécurité fonctionne pour un bot simple ; cependant, si une personne télécharge notre jeu et inspecte les en-têtes des requêtes envoyées depuis son ordinateur lorsqu'il joue, elle pourrait contourner cette sécurité. À ce jour, nous ne nous sommes pas encore consacrés davantage à cette partie.

2.2 Structure du jeu

2.2.1 Structure globale

Après avoir développé le système de multijoueur permettant d'envoyer des informations en temps réel, nous avons pu nous consacrer au développement du gameplay de notre jeu. Maintenant que nous disposons d'objets en C# représentant les éléments de notre jeu, nous pouvons développer nos algorithmes et le gameplay en utilisant ces derniers. Voici une représentation graphique des classes principales de notre jeu :

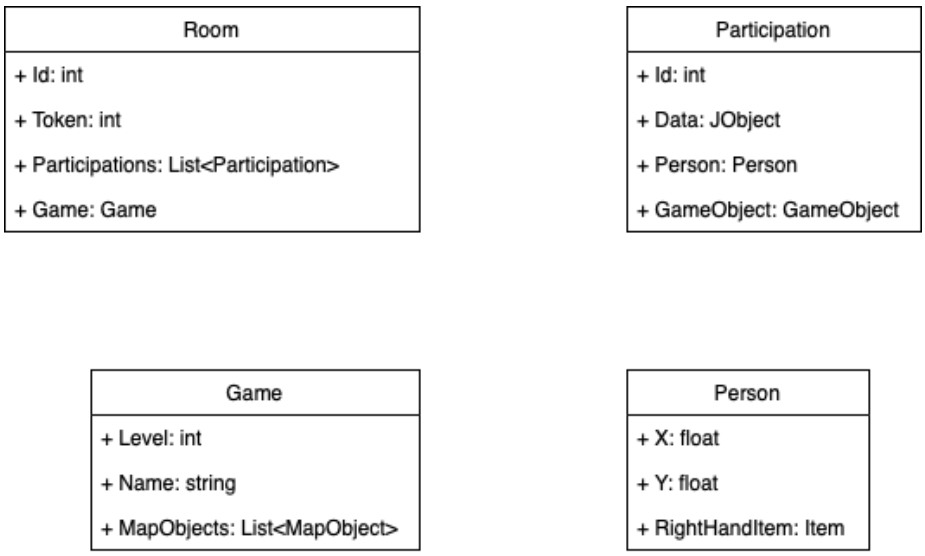

Figure 3: Représentation des différentes classes

De nombreuses classes enfants sont également présentes, mais elles ne sont pas nécessairement utiles à la compréhension de la structure générale du jeu. En effet, plusieurs autres interfaces et classes sont présentes afin de mieux cerner les caractéristiques de certains objets.

2.2.2 Construction d'une partie

Lorsque le joueur crée une partie de niveau 1, la classe Level1, héritant de la classe Game, est instanciée. Cette classe permet de définir les caractéristiques spécifiques de la construction du premier niveau (Le Petit Chaperon rouge). Une fois l'objet instancié, il est sérialisé au format JSON afin d'envoyer toutes les informations au serveur. Cela permet aux prochains joueurs qui rejoindront la partie de recevoir toutes les données nécessaires à leur expérience de jeu.

Si un autre joueur rejoint cette partie, il récupère l'ensemble des informations sous forme de JSON, puis les désérialise en un objet Level1, comme prévu dans l'architecture du jeu.

La partie a pour objectif de centraliser l'ensemble des éléments constitutifs du jeu. En effet, tous les objets et actions possibles sont accessibles depuis cette classe et ses attributs.

2.2.3 Gameplay

Le gameplay constitue la face visible du projet pour l'utilisateur. En effet, même un projet bien structuré et correctement architecturé ne suffit pas à convaincre un joueur de rester. Sans un gameplay attrayant, ce dernier n'aura aucun intérêt à rester dans le jeu et à continuer à l'explorer. Il est donc essentiel de consacrer du temps non seulement à la conception de l'architecture du jeu, mais aussi au développement des mécaniques de jeu et des actions réalisables par les joueurs.

Grâce à la structure actuelle, nous avons la possibilité d'accéder facilement aux attributs et aux méthodes de la partie ainsi qu'aux objets qui la composent. Par exemple, la méthode " Item.IsNearby(Participation) " permet de vérifier si le joueur courant est proche d'un objet sur la carte. Pour un coffre, cette méthode récupère le premier objet qui s'y trouve. Dans le cas de la maison, elle marque la réussite du niveau pour le Petit Chaperon rouge. Chaque joueur peut ainsi interagir avec différents objets comme les coffres, et s'il y a des items disponibles, il peut les récupérer. Il est également possible pour le joueur de se déplacer et d'explorer la carte.

Figure 4: Photo du jeu issue d'une partie en multijoueur

2.2.4 Intelligence Artificielle

L'intelligence artificielle est une partie complexe de notre jeu. En effet, nous souhaiterions l'implémenter dans les mouvements et les actions de certains personnages. Par exemple, le loup dans le conte du Petit Chaperon rouge. Le problème est que si nous voulons faire bouger un loup, alors tous les autres joueurs de la partie devront voir ces mêmes déplacements. Mais qui calcule les déplacements du loup ? Si un client s'occupe du calcul des déplacements, alors s'il quitte la partie, il n'y aura plus de calculs. Pour faire face à ce problème, nous avons trouvé deux solutions. La première est de choisir le premier participant de la partie comme étant celui qui va calculer ces déplacements. Cependant, cette technique pose un problème : ce client effectuera plus de requêtes que les autres, et s'il perd sa connexion à internet, le loup arrête de se déplacer pour les autres. La seconde technique est de définir les mouvements et les actions du loup depuis le serveur. Pour ce faire, un CRON serait mis en place pour effectuer une tâche sur l'API à intervalle de temps régulier. L'API récupérerait toutes les parties existantes et, pour chacune d'entre elles, regarderait les données et les participations qui y sont associées. Si un joueur est à proximité d'un loup, alors l'API enverrait un message via Pusher à tous les clients pour informer que le loup va commencer à se battre. Cette technique a l'avantage d'être totalement indépendante et de continuer à faire bouger le loup dans tous les cas. Cependant, le problème de cette idée est que les algorithmes de déplacements et d'actions du loup devraient être réalisés en PHP, le langage de notre API. Sachant que nous devons développer notre jeu en C#, nous allons probablement abandonner cette idée pour mettre en place la première. Pour pallier les désavantages de la première technique, nous allons, lors de la création de chaque participation, compter le temps que la requête a pris pour partir et revenir et ainsi voir qui a la connexion la plus rapide. Cela assignera donc celui qui a eu le meilleur résultat lors du test comme étant chargé de faire les calculs pour l'intelligence artificielle. Si ce premier joueur se déconnecte de la partie, il sera nécessaire de transmettre les responsabilités des calculs à un autre client. Un problème persiste cependant. Si le joueur chargé du calcul de l'intelligence artificielle perd sa connexion internet, comment informer le serveur qu'il ne peut plus s'en occuper ? Plusieurs possibilités s'offrent à nous : la première est d'envoyer une requête à intervalle de temps régulier vers ce client pour savoir s'il est toujours connecté, la seconde, la plus simple à mettre en place, est de ne pas se soucier de ce point de vue, en considérant que le joueur ne perde pas sa connexion.

2.2.5 Menu

Un domaine dans lequel nous avons travaillé est celui des menus du jeu. Les menus sont nécessaires pour aiguiller le joueur lors de son expérience sur la plateforme. Ainsi, lorsqu'on lance le jeu, une première page s'ouvre avec la possibilité de créer ou de rejoindre une partie. Un champ est alors visible afin de pouvoir renseigner le jeton d'authentification d'une partie déjà en cours, accompagné par un bouton pour la rejoindre. En dessous, la seconde option est

présente, celle de créer une nouvelle partie. À terme, nous souhaiterions avoir une liste d'images, chacune représentant un niveau du jeu. Ainsi, le joueur pourra cliquer sur l'une des images afin de jouer à ce niveau.

2.2.6 Graphismes

Les graphismes sont essentiels dans un jeu pour une expérience de jeu fluide et agréable. Nous avons donc décidé d'introduire notre personnage sous forme d'animation. Ce personnage n'est plus représenté par une seule image statique, mais par une succession de 7 images qui créent une animation en se répétant à l'infini. Nous avons également décidé d'ajouter des éléments de décor à notre carte. Pour ce faire, nous avons cherché différentes images qui conviennent à l'ambiance de notre jeu. Nous les avons ensuite ajoutées dans les différents composants de notre TileMap, qui nous permet de dessiner nos cartes de jeu à l'aide d'outils graphiques. Ces éléments sont essentiels, car ils sont généralement les premières choses observées par le joueur, qui se fera rapidement un premier avis sur le jeu. Soigner ces détails est donc primordial.

2.2.7 Site web

Un site web présentant l'intégralité de notre projet est disponible sur internet à l'adresse suivante : the-grimms.com. Sur celui-ci, nous présentons en détails l'ensemble des membres du groupe, les tâches de chacun ainsi que l'histoire de notre équipe. Il est également possible de télécharger l'ensemble des comptes rendus ainsi que la première version de notre jeu. Ce dernier est composé de quatre pages. La première est celle par défaut, elle met en avant un texte présentant l'histoire et la finalité de notre jeu. La seconde page met en avant les membres du groupe avec une brève description et les tâches auxquelles sont associées chacun des membres. La troisième page est celle montrant l'avancement de notre projet et les événements passés importants. Enfin, la dernière page est celle permettant de télécharger le rapport de soutenance. L'ensemble du site web a été développé en PHP 8. Une base de données y a été créée afin de pouvoir ajouter des données de manière dynamique. En effet, sur la page de présentation des membres, chaque personne correspond à un élément de la table User. Ainsi, le prénom, le nom, une image et une brève description sont renseignés à propos de chacun. Cette méthode est également utilisée pour enregistrer les différents éléments de la timeline, c'est-à-dire la liste des événements qui ont eu lieu jusqu'aujourd'hui. Développer un site web peut être une étape fastidieuse et répétitive pour tous. De nombreuses failles de sécurité peuvent apparaître rapidement et sont assez longues à corriger. C'est pour ces raisons que nous avons fait le choix d'utiliser le Framework PHP Symfony 7. L'utilisation de Frameworks s'est largement démocratisée ces dernières années. En effet, ceux-ci permettent d'utiliser des modules et composants développés par une communauté de développeurs, souvent bénévoles. Le choix de se tourner vers Symfony et pas un autre Framework s'est fait en raison des nombreuses ressources disponibles gratuitement en ligne. En effet, presque tous les problèmes auxquels nous pouvons faire face ont déjà une solution sur des forums et autres sites d'apprentissage. De plus, sa documentation bien développée nous permet d'avancer rapidement dans la conception de notre site web. Symfony 7 est un Framework dit ORM (Object-Relational Mapping). Ce mode de structure permet de créer une correspondance entre un objet et un modèle relationnel de base de données. De nombreuses méthodes sont à notre disposition pour nous permettre de faire des requêtes SQL de manière simplifiée. Le Framework suit une architecture précise et complexe à prendre en main. Tout le code côté serveur, dit backend, se situe dans le dossier src. Celui-ci est composé de différents sous-dossiers tels que pour la gestion des tables de la base de données et les Controllers qui sont la logique de notre site. Pour tout le rendu côté client, c'est-à-dire le HTML et la mise en forme graphique de notre site, l'ensemble des ressources se situent dans le dossier Template. De nombreux autres dossiers et fichiers sont présents pour la configuration et la gestion de nombreux éléments composant notre site. Après un premier développement de notre projet en local sur nos ordinateurs, nous avons fait le choix de le publier sur internet. Pour cela, nous avons décidé de l'héberger sur un serveur dont nous avions déjà accès avant le début de notre projet. Après avoir configuré notre site avec les informations du serveur, nous avons désormais une base de

données en production reliée au site web fonctionnel.

2.3 Les problèmes rencontrés

2.3.1 Multijoueur (sérialisation, désérialisation)

L'un des problèmes qui nous a pris le plus de temps est celui de la sérialisation et de la désérialisation des informations. En effet, ne connaissant pas ces méthodes en C#, nous n'envoyions que les valeurs x et y, ce qui ne posait alors aucun problème. Cependant, dès que nous avons voulu synchroniser une partie entière, nous avons dû chercher des solutions. Nous sommes alors tombés sur le principe de sérialisation et de désérialisation intégré directement dans C#. Nous avons donc fait le choix de l'utiliser, car il permet de transformer rapidement et efficacement des objets en JSON. Après avoir été bloqués pendant de nombreuses heures sur ce sujet, nous avons finalement trouvé une solution.

2.3.2 Utilisation de l'outil Git

La collaboration entre les différents membres du groupe est difficile à gérer. En effet, après avoir réfléchi ensemble sur les idées de notre jeu, il est nécessaire de trouver un moyen pour collaborer dans l'avancée du projet. Pour ce faire, nous avons utilisé le repository GitLab qui nous a été mis à disposition pour avancer sur le projet. Cependant, la prise en main de cet outil est complexe à comprendre et à maîtriser. En effet, si tous les membres envoient leur code sur la branche principale, de nombreux conflits apparaissent s'ils ont travaillé sur les mêmes fichiers. Pour pallier ce problème, nous utilisons le système de branches mis à disposition par Git. Nous avons choisi de structurer notre repository de la manière suivante : la branche master, qui permet d'avoir une version finale de notre projet pour chaque soutenance, et une branche dev, qui regroupe l'ensemble des avancées dans le projet. Lorsqu'un collaborateur veut ajouter une fonctionnalité au jeu, il crée une branche feature-x à partir de la dernière version de la branche dev. Lorsque ce dernier a terminé ses changements, il envoie sa branche vers la branche dev du repository. Ainsi, sa nouvelle branche devient la dernière version de dev. Lorsque nous avons ajouté toutes nos modifications, nous faisons un seul et unique push de la branche dev vers master. L'ensemble de ces techniques est difficile à prendre en main et nécessite de la pratique. Nous avons, à de multiples reprises, perdu de nombreux fichiers sur lesquels nous avions travaillé. Heureusement, en cherchant dans les logs du repository, il est possible de récupérer la version qui nous convient et de la déplacer vers la dernière version que nous voulons. Ainsi, à partir du moment où nous commitons, nous pouvons retrouver tout travail perdu.

3 Ce que nous allons réaliser

3.1 Design des cartes, assets animés

De nombreuses tâches sont encore à réaliser avant de considérer notre projet comme terminé. En effet, il nous reste à travailler dans différents domaines afin de rendre le jeu plus dynamique, plus attrayant et agréable à jouer.

Tout d'abord, nous souhaiterions modifier plusieurs images composant l'ambiance de notre jeu. En effet, nous aimerions rendre l'ensemble de nos composants animés. Pour cela, il nous sera nécessaire de dessiner de nombreuses images pour chaque personnage ou chaque objet afin de pouvoir en faire une animation, ou bien trouver des " assets " correspondant à nos attentes sur Internet. Actuellement, tous nos dessins étant statiques, mis à part le personnage, rendent le jeu moins attrayant. En ajoutant des animations sur les arbres, le coffre, la maison et tous les autres éléments, cela rendrait le jeu beaucoup plus vivant.

3.2 Ajout d'objets

Maintenant que nous disposons d'une structure de jeu dynamique, il nous est plus facile d'ajouter de nombreux objets en jeu. En effet, si nous prenons l'exemple du coffre, il s'agit uniquement d'un objet dépendant de la classe abstraite MapObject, qui regroupe tous les objets composant notre carte. Si nous voulons ajouter de nouveaux objets, tels qu'une maison ou un arbre magique, il suffira d'ajouter une instance de cet objet dans une liste. L'ajout d'autres objets rendra la partie plus longue à jouer. En effet, cela nécessitera au joueur de chercher ces objets et d'y effectuer des actions, ce qui permettra d'ajouter du contenu.

3.3 Ajout de techniques de jeu

Afin de rendre le jeu plus intéressant, il nous est nécessaire d'ajouter de nombreuses techniques de jeu. Une technique de jeu consiste en la réalisation d'une action qui aura une conséquence dans le jeu. Par exemple, se battre, ramasser et poser des objets, etc. Tout cela fait partie du gameplay de notre jeu. Il s'agit donc d'un développement à long terme, car il représente la ligne directrice qui permettra au joueur de déterminer s'il aime ou non notre jeu.

Pour cela, nous avons différentes idées. Tout d'abord, nous aimerions mettre en place un système de combat contre un loup dans le conte du Petit Chaperon rouge. Ainsi, nous placerons un personnage avec des attributs spécifiques. Lorsque le joueur sera à proximité de sa cible, il pourra presser une touche afin de donner un coup. Face à lui, le loup en donnera également. Ainsi, les deux personnages perdront des points de vie.

3.4 Intelligence artificielle

Après avoir réfléchi à la manière de procéder aux calculs de l'intelligence artificielle, nous devons faire le choix de l'implémenter à travers quelque chose de visuel. Pour cela, les actions et déplacements du loup seront gérés par la partie intelligence artificielle de notre jeu.

En ce qui concerne le combat, nous avons l'idée d'utiliser la fonction " IsNearby() " qui nous permettrait de savoir si un joueur est à proximité du loup. Une fois le joueur proche du loup et s'il n'est déjà pas en combat, alors le combat commencera. Pour ce faire, un message sera envoyé à l'API pour informer que le loup est en combat et qu'aucun autre joueur ne pourra en initier un. Ainsi, les calculs des actions du loup se feront sur l'ordinateur du client. Cela décidera de ses prochains coups ainsi que de ses déplacements.

3.5 Optimisations de code

Afin de rendre notre jeu plus rapide, fluide et pour éviter tout problème, il nous est nécessaire d'optimiser notre code. Cela peut se faire sur différentes parties. Tout d'abord, lorsque nous créons, par exemple, des fonctions qui permettent de calculer certaines choses, il est nécessaire de voir s'il existe d'autres méthodes de calcul.

Ensuite, une grande partie de l'optimisation du multijoueur peut être réalisée. En effet, lorsqu'un joueur bouge dans la partie, il envoie l'intégralité des informations le concernant. Bien que cela ne représente que très peu de données, ce sont toujours des données superflues. Le véritable problème ne vient pas de là, mais de la manière dont nous traitons ces données. En effet, lorsqu'un client reçoit un message avec de nouvelles données d'un joueur, telles que de nouvelles coordonnées, le client supprime l'ancien préfabriqué concernant ce joueur et en instancie un nouveau.

Une solution intéressante serait de remplacer uniquement les coordonnées X et Y, cependant un problème persiste. Si le joueur prend une épée en main, alors la représentation graphique change également. Nous sommes donc contraints d'instancier un nouveau préfabriqué. Pour résoudre ce problème, plusieurs possibilités s'offrent à nous. La première serait de vérifier s'il est nécessaire de changer son apparence. Si cela n'est pas utile, alors on modifie uniquement son vecteur de position ; sinon, on instancie un nouveau préfabriqué. Une autre idée serait de transformer les différents préfabriqués (chaperon, chaperon avec épée) en un seul et unique préfabriqué : le chaperon. Pour ajouter une épée dans la main, il serait nécessaire de créer une variante de ce dernier afin de lui ajouter des attributs graphiques. Cela permettrait d'optimiser les calculs des ordinateurs des clients, mais serait assez complexe à mettre en place.

Pour optimiser notre code, il est également nécessaire de développer des fonctions qui seront utilisées de nombreuses fois. En effet, lorsque l'on rejoint ou modifie les informations d'un joueur, on effectue la même tâche, mais certaines actions sont séparées. Un grand travail de nettoyage et d'optimisation serait nécessaire pour plus de clarté dans notre travail.

Il est important de peser le pour et le contre de la simplification du code, quitte à perdre en optimisation des ressources, ou inversement.

4 Conclusion

Pour conclure, après des dizaines d'heures de travail, nous sommes ravis de commencer à voir notre projet prendre vie. En effet, avec l'intégration de notre API, de notre système multijoueur ainsi qu'une bonne architecture du projet, nous pouvons enfin voir les résultats de ce travail. Depuis la naissance de notre projet, nous avons dû faire face à de très nombreux problèmes, tant sur le plan logistique que technique. De nombreuses solutions intéressantes ont été développées. La structure de ce dernier peut sembler fastidieuse, mais elle est en réalité très intéressante et efficace. Cependant, de nombreuses choses restent encore à développer. En effet, une grande partie du gameplay n'est pas terminée et nécessite que nous y consacrions davantage de temps. Nous sommes certains que l'ensemble de nos niveaux plaira à nos futurs joueurs et que les contes des frères Grimm tiendront en haleine toute personne installant notre jeu.