Once Upon a Time

Alexandre LEDOUX
Lucas MOREL
Mantas KRUKONIS
Nicolas SIMEONOV

Sommaire

1 Présentation du projet

1.1 Introduction

Animé par une réelle volonté de transmettre la passion des contes des frères Grimm au plus grand nombre, le studio The Grimms a décidé de se lancer dans le développement d'un jeu vidéo. L'objectif est de permettre à chacun de découvrir ou redécouvrir ces fabuleux contes de manière ludique et immersive.

Cette idée est née d'un constat simple : de plus en plus de jeunes consacrent leur temps libre aux jeux vidéo, au détriment de la lecture. Nous avons donc voulu créer un jeu dans lequel chaque joueur incarne le personnage principal d'un conte, alliant ainsi le plaisir du jeu à la richesse culturelle des récits des frères Grimm.

1.2 Présentation de notre jeu

Pour maintenir l'intérêt du joueur tout au long de l'expérience, il est essentiel de proposer plusieurs mécanismes qui éveillent sa curiosité et son envie de continuer à jouer. C'est pourquoi nous avons choisi de structurer notre jeu sous forme de niveaux indépendants, chacun représentant un conte différent. Cette approche permet au joueur de s'immerger dans un univers unique.

Actuellement, notre jeu inclut le conte du Petit Chaperon rouge, avec des éléments spécifiques qui lui sont propres. L'objectif est simple : vous incarnez le Petit Chaperon rouge et devez retrouver la galette que votre mère vous a préparée pour l'offrir à votre grand-mère, dont la maison est perdue au cœur de la forêt. Sur votre chemin, il faudra éviter les loups et tous les ennemis susceptibles de vous nuire.

Pour rendre l'aventure encore plus captivante, notre jeu propose un mode multijoueur, vous permettant de jouer avec vos amis et de coopérer pour atteindre ensemble votre but. Durant votre quête, vous découvrirez de nombreux objets utiles que vous pourrez collecter et utiliser au fil de la partie.

1.3 Fonctionnement d'une partie

Lorsqu'un joueur arrive sur notre jeu, il peut choisir soit de créer une nouvelle partie, soit de rejoindre une partie en ligne déjà existante. S'il crée une partie, il sélectionne alors le niveau sur lequel il souhaite jouer. S'il décide de rejoindre une partie, il récupère toutes les informations en temps réel : la position des autres joueurs, l'emplacement des coffres, les points de vie de chacun, etc.

Pendant la partie, tous les joueurs doivent coopérer pour explorer l'environnement, trouver des objets, dénicher des passages secrets et obtenir des avantages qui les aideront à atteindre l'objectif du niveau. Par exemple, dans le conte du Petit Chaperon rouge, la mission finale consiste à accéder à la maison de la grand-mère.

La partie se termine lorsque l'un des joueurs réussit à apporter la galette à sa grand-mère. Dans ce cas, tous les joueurs remportent ensemble le niveau, récompensant leur collaboration et leur persévérance.

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 constitue l'intégralité de notre jeu. En combinant tous ces développements, nous obtenons le produit final attendu.

Pour respecter nos échéances, il a été essentiel de travailler régulièrement sur le développement. Une répartition claire et efficace des tâches, telle que présentée dans le cahier des charges technique, a également été indispensable.

Certaines tâches demandent plus de temps que d'autres, ce qui justifie les différences dans notre planning. En particulier, le développement du multijoueur et du gameplay représentent les piliers du projet. Sans ces éléments, notre jeu ne pourrait pas exister. C'est pourquoi nous avons décidé de consacrer la majeure partie de notre temps de développement à ces deux aspects.

À l'heure actuelle, l'ensemble des échéances concernant les 9 principaux domaines de développement est respecté pour la livraison du projet.

Comme nous pouvons le constater, le site web a été entièrement développé et est actuellement fonctionnel à l'adresse suivante : the-grimms.com. Le gameplay de notre jeu est terminé : de nombreux objets, interactions et mécaniques ont été ajoutés afin de rendre l'expérience de jeu plus dynamique.

Le système multijoueur a été mis en place, permettant de jouer seul ou à plusieurs sur une même carte. L'intelligence artificielle a été améliorée grâce à l'implémentation de l'algorithme de Dijkstra ainsi que d'autres méthodes de calcul.

Nous avons également créé des menus pour guider le joueur tout au long de son expérience. Les graphismes ont été perfectionnés afin de rendre l'expérience utilisateur plus agréable.

L'interface a été peaufinée, notamment avec l'ajout d'une barre d'informations présente en jeu. Aucun son ni musique n'a été intégré, suite au départ précipité de notre seul musicien. Nous avons donc choisi de ne pas inclure de bande-son.

Enfin, de nombreux tests ont été réalisés tout au long de la période de développement, mais aussi en phase finale, afin de vérifier que le jeu fonctionne correctement sur toutes les plateformes ciblées.

Avec tous les développements réalisés, nous avons donc réussi à livrer notre produit final en temps et en heure, tout en respectant intégralement le cahier des charges.

2 Système de multijoueur

2.1 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.

2.2 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.

2.3 Présentation de Pusher

Pour assurer la synchronisation en temps réel, le client envoie ses nouvelles données au serveur, qui doit ensuite les diffuser à tous les autres clients connectés à la même partie. Il est facile d'envoyer une réponse au client à l'origine de la requête, mais informer les autres clients est plus complexe : sans solution dédiée, ceux-ci devraient interroger régulièrement le serveur pour vérifier s'il y a du nouveau, ce qui serait très coûteux en ressources.

Pour éviter ce problème, nous avons opté pour un système de WebSocket. Ce protocole utilise un serveur Node.js capable d'envoyer des informations directement du serveur aux clients connectés à un canal spécifique. Ne souhaitant pas développer et maintenir notre propre serveur Node.js, nous avons

choisi d'utiliser la plateforme Pusher. Cette solution nous permet d'envoyer des requêtes via leur API en précisant trois éléments : le canal (channel) concerné, l'événement (event) à transmettre, et les données à envoyer. Pusher se charge alors de distribuer ces informations à tous les clients abonnés à ce canal, c'est-à-dire nos joueurs en ligne.

La plateforme propose un plan gratuit avec une limite de 200 000 messages par jour, ce qui est largement suffisant pour notre usage.

La structure de notre système multijoueur est illustrée par la figure 5, avec notre API, celle de Pusher, et trois clients représentant chacun un joueur connecté à la même partie.

2.4 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 accepte des requêtes en POST et prend en paramètre le type d'événement à traiter, parmi les suivants :

event-newParticipation : ajout d'un nouveau participant event-participation : mise à jour des informations d'un participant event-room : mise à jour des informations d'une Room event-disconnect : déconnexion d'un joueur event-

wolf : nouveaux points cibles d'un loup après calcul de son itinéraire via l'algorithme de Dijkstra event-finished : fin du niveau par un joueur présent dans la partie L'API reçoit aussi l'ID de la Room ou de la Participation concernée, ainsi que les data, c'est-à-dire les nouvelles données au format JSON à appliquer.

En fonction de l'événement reçu, l'API agit de la manière suivante :

Pour un nouveau participant, elle envoie à Pusher un message avec l'événement correspondant et les données du participant. Pour une mise à jour d'une participation ou d'une Room, elle modifie le champ data dans la base de données, puis envoie les nouvelles informations à Pusher avec l'événement dédié. En cas de déconnexion, elle supprime la participation concernée de la base de données. Si le joueur était le dernier dans la Room, celle-ci est également supprimée. Ensuite, elle informe Pusher de ce changement.

2.5 Salons et événements Pusher

Pour permettre la gestion simultanée de plusieurs parties multijoueurs sans interférence entre les messages, nous avons décidé de créer des salons distincts pour chaque partie. Chaque salon peut gérer plusieurs événements différents. Nous avons donc instancié pour chacun des salons les six événements présentés précédemment.

Lorsqu'un client envoie une requête au serveur, il n'est pas nécessaire de spécifier explicitement le salon Pusher correspondant à la partie. En effet, en transmettant uniquement l'ID de la partie en cours, le serveur peut, grâce à une requête SQL, retrouver ce salon dans sa base de données. Ainsi, à la réception d'une requête de diffusion d'un message Pusher, le serveur crée un nouveau message destiné à Pusher, en lui indiquant de diffuser ce message dans le salon nommé "room-id" (où id correspond à l'ID de la partie), sur l'événement spécifique indiqué dans le corps de la requête.

Ce système efficace garantit que les messages sont envoyés uniquement aux joueurs concernés, c'est-à-dire ceux présents dans la même partie.

2.6 Transformation des objets C# en JSON et inversement (Sérialisation, Désérialisation)

Dans notre code C, toutes les entités sont représentées sous forme d'objets. Cette approche facilite l'utilisation d'attributs et de méthodes communes, tout en assurant un code clair et optimisé. Cependant, pour transmettre les informations d'un objet vers notre API, il est nécessaire de convertir cet objet en format JSON, un processus appelé sérialisation. Une méthode C permet d'effectuer cette opération facilement.

Toutefois, certains attributs ne peuvent pas être sérialisés. Par exemple, dans notre objet Participation qui représente un joueur dans la partie — l'attribut GameObject de Unity, responsable de l'affichage du joueur dans le jeu, contient de nombreuses données complexes rendant sa sérialisation impossible.

Ainsi, lors de l'envoi des données, nous ne transmettons pas cet attribut GameObject. Cela n'est pas problématique, car les autres clients peuvent recréer localement cet objet à partir des autres informations reçues, telles que la position (`x, y`) ou l'image de représentation.

Lorsqu'un événement est déclenché par le serveur, par exemple pour mettre à jour les informations d'une participation, nous recevons les données sous forme de JSON. Nous devons alors associer ces données au joueur concerné. Pour ce faire, nous désérialisons le JSON, créons un nouveau GameObject pour la participation, puis positionnons cet objet à l'endroit qu'il occupait précédemment dans le jeu.

Ce processus de sérialisation et désérialisation est essentiel au bon fonctionnement de notre jeu. Grâce à des méthodes simples et efficaces, nous pouvons mettre à jour les informations rapidement en appelant une fonction unique, ce qui accélère significativement le développement du gameplay.

2.7 Présentation de la création d'une partie

Lorsqu'un joueur souhaite commencer une nouvelle partie, il clique sur le bouton Nouvelle partie et sélectionne un niveau. S'il choisit le niveau 1, correspondant au conte du Petit Chaperon rouge, cela crée en C un objet Level1, qui contient des informations telles qu'une liste d'objets (coffres, maisons, etc.) ainsi que des données par défaut, comme les coordonnées x et y attribuées à tous les nouveaux participants. Une fois cet objet Level1 instancié, héritant de la classe Game, le client sérialise toutes ces informations en JSON et les envoie via une requête POST pour créer la Room. Ainsi, toutes ces données communes à tous les joueurs sont synchronisées sur le serveur.

2.8 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.

2.9 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.

2.10 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.

2.11 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.

2.12 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.

2.13 Calcul autonome en multijoueur

L'intelligence artificielle est une partie complexe de notre jeu. En effet, souhaitant 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 le premier joueur chargé des calculs se déconnecte de la partie, il devient nécessaire de transférer la responsabilité de ces calculs à un autre client. Cependant, un problème subsiste : si ce joueur perd sa connexion internet, comment le serveur peut-il être informé qu'il ne peut plus assurer cette tâche ?

Plusieurs solutions existent :

La première consiste à envoyer régulièrement des requêtes à intervalle de temps régulier au client pour vérifier s'il est toujours connecté. La seconde, plus simple à mettre en œuvre, est de ne pas se préoccuper de ce cas, en supposant que le joueur ne perd pas sa connexion. Pour garder un code simple, clair et facile à déboguer, nous avons choisi la deuxième option.

Ainsi, lorsqu'un joueur s'approche d'un loup et commence à interagir avec lui, une requête de mise à jour de l'état du loup est envoyée à tous les participants. Chaque client reçoit ces informations et exécute de manière autonome la logique de déplacement et d'action du loup.

Cette approche décentralisée permet à chaque client d'être responsable de l'intelligence artificielle du loup localement, évitant ainsi tout point unique de défaillance lié à la connexion d'un joueur particulier.

2.14 Problèmes de latence

Lors de notre seconde soutenance, le jury nous a interrogés sur notre système multijoueur. En effet, lors de la démonstration, lorsque deux joueurs jouaient simultanément dans une même partie, des téléportations soudaines apparaissaient régulièrement lors des déplacements des personnages. Ce phénomène était dû au fait que lorsqu'un joueur se déplaçait, il envoyait une requête toutes les 50 millisecondes afin d'éviter de surcharger le serveur.

Lorsque les autres joueurs connectés à la même partie recevaient les nouvelles coordonnées du joueur en mouvement, ils mettaient à jour localement la position du personnage, provoquant ainsi un effet de téléportation peu naturel à l'écran. Ce système, bien que fonctionnel, dégradait nettement l'expérience utilisateur.

Pour résoudre ce problème, nous avons réfléchi à deux solutions possibles. Actuellement, les déplacements sont envoyés via le protocole TCP, qui a l'avantage d'être fiable : il garantit la livraison des paquets dans le bon ordre, sans perte ni duplication. Cependant, cette fiabilité se fait au détriment de la rapidité, ce qui n'est pas optimal pour un jeu multijoueur en temps réel.

L'autre protocole envisageable est l'UDP, qui offre une meilleure rapidité car il ne comporte pas les mêmes contrôles que TCP. Cependant, UDP est moins fiable, puisqu'il ne garantit ni la livraison ni l'ordre des paquets.

Après avoir étudié ces deux options, nous avons finalement décidé de conserver notre méthode actuelle basée sur TCP, afin de ne pas restructurer entièrement notre système multijoueur, malgré les limites observées.

Nous avons donc choisi d'opérer des changements au niveau logiciel du jeu, plutôt que sur la partie réseau. Voici comment nous avons procédé :

Auparavant, lorsqu'un joueur se déplaçait, tous les autres clients recevaient ses nouvelles coordonnées et téléportaient directement son personnage à cette position.

Pour améliorer cela, nous avons ajouté, localement sur chaque client, un champ appelé TargetPoint pour chaque joueur présent dans la partie. Ce champ mémorise un point contenant des coordonnées X et Y. Ainsi, lorsqu'une nouvelle position est reçue, le TargetPoint est mis à jour avec ces nouvelles coordonnées, et le préfabriqué représentant le joueur initiant le mouvement subit une transformation progressive.

En parallèle, une fonction est appelée à chaque frame. Pour chaque joueur dont la distance entre sa position actuelle et son TargetPoint est supérieure à 0,01, cette fonction exécute Person.UpdateCoord(). Cette méthode déplace alors progressivement le personnage d'un certain nombre de pixels vers sa destination finale.

Comme cette opération est répétée à chaque frame, cela crée une impres-

sion de déplacement fluide et naturel, sans à-coups ni téléportations instantanées.

Cette méthode nous permet également de réduire le nombre de requêtes HTTP envoyées, ce qui diminue la charge sur le serveur tout en améliorant considérablement l'expérience utilisateur.

3 Intelligence artificielle

3.1 Introduction de l'intelligence artificielle

Afin de répondre aux exigences du cahier des charges, nous avons dû développer certains algorithmes intelligents plus élaborés. Pour ce faire, nous avons décidé d'intégrer notre système d'intelligence artificielle dans les déplacements des animaux. Autrement dit, lorsque l'on s'approche d'un loup par exemple, celui-ci réagit en nous suivant afin de donner des coups. Cependant avec la version précédant de l'algorithme de déplacement du loup, celui-ci réalisait une ligne droite jusqu'à sa cible, en passant à travers les arbres, les bâtiments et les objets. Le loup coupait donc tous les chemins et contraintes présentent sur le jeu. Afin de corriger le problème, nous avons décidé d'introduire l'algorithme de Dijkstra afin que les animaux se déplacent uniquement sur les chemins prédéfinis et non en ligne droite.

3.2 Premier algorithme

La première idée que nous avons explorée pour notre algorithme a été d'utiliser des rectangles invisibles, positionnés grâce à leurs coordonnées X et Y en haut à gauche et en bas à droite de chaque obstacle. Pour que l'algorithme fonctionne de manière optimale, il était nécessaire de placer ces rectangles sur tous les obstacles présents sur la carte. L'objectif était ensuite de déplacer le loup en ligne droite vers le joueur, mais si un obstacle se trouvait sur son chemin, il devait contourner cet obstacle soit par la gauche, soit par la droite, en choisissant la distance la plus courte, tout en continuant à vérifier si l'obstacle restait présent devant lui.

Cette méthode présente l'avantage d'être relativement simple à développer et ne nécessite pas d'algorithme complexe. Cependant, elle souffre de plusieurs inconvénients majeurs. Premièrement, il faut cartographier précisément tous les obstacles, ce qui implique un travail fastidieux et une maintenance lourde à chaque modification graphique de la carte. Deuxièmement, si cette cartographie est incomplète ou imprécise, les déplacements des animaux deviennent erratiques et peu naturels.

Face à ces limites, nous avons finalement opté pour un algorithme de pathfinding plus robuste et reconnu : l'algorithme de Dijkstra.

3.3 Présentation de l'algorithme de Dijkstra

L'algorithme que nous avons choisi répond au mieux aux problèmes rencontrés grâce à sa simplicité de développement et sa fiabilité. Son fonction-

nement est relativement simple.

Tout d'abord, il faut créer un graphe composé de points reliés entre eux. Pour cela, nous avons conçu une classe appelée Point , qui possède un nom, des coordonnées X et Y, ainsi qu'un dictionnaire listant ses points voisins. Le graphe est donc constitué de ces différents points, chacun connecté à ses voisins avec une distance associée.

Pour calculer la distance entre deux points, nous utilisons la formule classique de la distance euclidienne dans un repère 2D. Chaque point représente une intersection de chemins, ce qui permet au loup de se déplacer uniquement sur les chemins en terre.

Comme illustré sur la figure 1, nous pouvons observé un graphe. Chaque point correspond à une intersection de chemins dans notre jeu, permettant ainsi aux loups de se déplacer uniquement sur les routes.

Les connexions rouges représentent l'ensemble du plus court chemin entre le point A et le point E. Pour atteindre le point E, le loup devra donc passer successivement par les points B, C et D.

Une fois le graphe construit, il est nécessaire d'obtenir la liste des points à suivre pour aller du point initial au point final.

L'algorithme se déroule ainsi : Nous partons du point initial et examinons l'ensemble de ses voisins. Pour ce point de départ, il n'existe pas d'itinéraire plus rapide que celui qui commence par lui-même, nous le marquons donc comme non réutilisable . Ensuite, tant qu'il reste des points utilisables, nous identifions le voisin le plus proche (celui qui minimise la distance cumulée depuis le départ). Nous répétons cette opération pour ce voisin, en le marquant également comme non réutilisable. Progressivement, nous avançons vers le voisin le plus proche en additionnant les distances depuis les points précédents.

Après avoir développé et validé l'algorithme de Dijkstra, nous avons dû l'intégrer dans la méthode de déplacement des animaux.

Pour rappel, le moteur Unity exécute la fonction Update en continu, pouvant tourner jusqu'à 500 fois par seconde. Étant donné que le loup est autonome, sa prise de décision peut potentiellement se faire à chaque frame, ce qui représente un coût important en termes de performance.

Pour intégrer l'algorithme dans le comportement du loup, nous avons dû l'ajouter au réseau de points. Pour cela, nous avons créé un point temporaire, connecté au point du graphe existant le plus proche de sa position. Cela permet de calculer un itinéraire optimal entre le loup et n'importe quel autre point du graphe.

Le même problème se pose du côté du joueur, qui n'est pas non plus positionné sur un point du graphe. Nous avons donc également créé un point temporaire pour le joueur, relié au point le plus proche. Ainsi, il devient

possible de calculer un plus court chemin entre le loup et sa cible.

Cependant, un problème persiste : le joueur peut continuer à se déplacer, tandis que le loup suit un itinéraire déjà calculé. Il est donc indispensable de recalculer régulièrement cet itinéraire. La question est alors : à quelle fréquence effectuer ce recalcul ?

3.4 Problème lors de l'implémentation de l'algorithme

En choisissant de recalculer l'itinéraire à chaque frame, ou même toutes les x secondes, un problème survient. Supposons que le loup doive suivre le parcours A $\rightarrow$ B $\rightarrow$ C et qu'il se trouve actuellement entre A et B, mais plus proche de A. Lors du recalcul, l'algorithme génère de nouveau l'itinéraire complet A $\rightarrow$ B $\rightarrow$ C, ce qui pousse le loup à retourner vers A. Cela provoque un aller-retour infini entre les points A et B.

La première solution envisagée a été d'ignorer ce premier point dans le calcul. Cependant, si le joueur s'est déplacé de l'autre côté de la carte, le loup risque alors de prendre un chemin coupant un point important, ce qui dénature le sens de l'algorithme et rend la trajectoire moins pertinente.

3.5 Version finale de notre intelligence artificielle

Pour intégrer l'algorithme de Dijkstra dans la fonction de décision des loups, nous avons décidé de le faire de la manière suivante ; Chaque loup possède un point cible appelé TargetPoint . Cette valeur peut être nul par défaut. Lorsqu'un joueur s'approche d'un loup, ce dernier va donc avoir ce joueur dans le viseur, jusqu'à sa mort. Ainsi, il va calculer une première fois l'itinéraire entre le loup et joueur, chacun étant connecte au point le plus proche. Le loup va donc calculer un premier itinéraire. A chaque frame, au lieu de recalculer l'itinéraire, on regarde si le loup est actuellement proche des coordonnées x et y de sa cible. S'il n'est pas proche, cela signifie que le loup est actuellement en train de se déplacer d'un point à un autre, et on le laisse terminer. Dans le cas contraire, le loup est proche de la cible (à moins d'une case), on va donc calculer son nouvel itinéraire jusqu'au joueur qu'il vise. Le loup étant proche de son premier point du trajet, sa nouvelle cible devient le deuxième point du trajet afin d'éviter une boucle infinie sur le même point. S'il n'existe pas de second point, cela signifie que le loup se déplace directement vers le joueur car le trajet est plus court qu'en passant par un point externe.

3.6 Conclusion sur l'intelligence artificielle

Avec l'implémentation de l'algorithme de Dijkstra dans notre jeu, les mouvements des animaux deviennent plus réalistes leur permet de suivre des chemins réels. Tout cela apporte une réelle plus-value au jeu car les animaux deviennent de réelles contraintes dans l'avancement vers la victoire.

4 Structure du jeu

4.1 Structure globale

Après avoir développé le système multijoueur permettant la transmission d'informations en temps réel, nous avons pu nous concentrer sur le développement du gameplay. Désormais, avec des objets c# représentant les différents éléments du jeu, il devient possible de faire interagir ces éléments entre eux. Voici une représentation graphique des classes principales :

Comme vous pouvez l'observer sur la figure 2, la classe Room est la pièce centrale de notre jeu. Elle joue le rôle de point d'entrée entre le serveur et le client. Toute action ou interaction avec les autres joueurs en ligne passe par cette classe. C'est pour cette raison que la classe Room possède une instance statique, permettant d'accéder à Room.Instance depuis n'importe quel fichier du jeu.

Ainsi, lorsqu'un joueur crée ou rejoint une partie, une instance de Room est créée côté client. Cette instance possède des caractéristiques uniques et des informations essentielles, jouant le rôle de chef d'orchestre sur l'ordinateur du joueur. Lors de son instanciation, elle se connecte aux événements WebSocket de la partie dans laquelle le joueur se trouve.

La classe Room instancie ensuite tous les autres joueurs, animaux et objets présents dans la partie. Dès la réception d'un message Pusher, elle met à jour les informations reçues en les transmettant aux classes qui lui sont connectées. Étant basée sur une interface de Unity, cette classe peut exécuter une fonction Update() à chaque frame du jeu, ce qui nous permet de gérer en temps réel les interactions et déplacements des éléments du jeu.

Afin de nous laisser la possibilité de développer plusieurs niveaux différents, nous avons créé une classe appelée Game. Cette classe possède divers attributs tels qu'un nom, une liste d'objets, de créatures ou de bâtiments positionnés à des coordonnées précises sur la carte. Elle contient également des coordonnées d'apparition et de fin de niveau.

Différents niveaux peuvent être développés en créant des classes Level qui héritent de la classe Game. Ainsi, chaque niveau hérite des propriétés de Game, ce qui nous permet de positionner précisément les éléments du jeu, de modifier les lieux d'apparition, et d'ajouter d'autres options spécifiques à chaque niveau.

Dans la room principale, chaque client possède une liste regroupant l'ensemble des autres joueurs présents dans la même partie.

La classe Participation représente un joueur au niveau réseau. Son id correspond à celui enregistré dans la base de données côté serveur, ce qui permet de l'identifier facilement.

L'attribut Data contient l'ensemble des informations du joueur sous forme de JObject, une structure proche du format JSON. Cette donnée est directement liée au champ JSON stocké en base de données. Elle regroupe toutes les informations importantes concernant le joueur dans le jeu, comme sa position, les objets qu'il tient en main, ses points de vie, etc.

Enfin, l'attribut Person est une copie de Data, mais désérialisée en un objet Person. Cet objet permet d'agir directement sur le joueur grâce à ses attributs et méthodes spécifiques, facilitant ainsi la manipulation des données du joueur dans le code.

Person est donc la représentation d'un joueur dans le jeu. Grâce à ses nombreux attributs, il correspond à ce que les joueurs voient graphiquement d'eux-mêmes et de leurs amis. Par exemple, lorsqu'un joueur se déplace, il met à jour ses coordonnées dans Participation.Person.X puis envoie ce changement via Room.SendMessage().

La classe Wolf représente les loups dans le jeu. Dans le conte du Petit Chaperon rouge, ce dernier tente de se rendre chez sa grand-mère, mais croise la route du loup. Ainsi, lorsqu'un joueur rencontre un loup, il se fait attaquer par ce dernier.

Chaque loup possède un id unique pour le distinguer des autres, un attribut FollowParticipation qui stocke l'ID du joueur qu'il doit suivre, ainsi qu'un booléen indiquant s'il est en train de combattre.

C'est également dans cette classe que l'algorithme de Dijkstra est implémenté pour calculer le chemin que le loup doit emprunter afin d'atteindre sa cible.

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.

Pour faciliter et accélérer le développement à long terme, nous avons créé deux fonctions dédiées à l'envoi de données aux autres participants dans le cadre du multijoueur : SendParticipationData() et SendRoomData().

Lorsque seul le personnage d'un joueur est impacté, par exemple lors d'un déplacement ou d'une perte de points de vie, c'est SendParticipationData() qui est appelée. En revanche, quand une action concerne l'ensemble des joueurs dans la partie — comme la prise en main d'un objet sur la carte ou l'attaque d'un loup — la fonction SendRoomData() est utilisée.

Chacune de ces fonctions exécute ensuite une requête vers l'API avec le nom précis de l'événement correspondant, ce qui permet aux autres joueurs de traiter correctement et efficacement l'information reçue.

4.2 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.

4.3 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.

4.4 Franchissement des obstacles

Lors de la soutenance précédente, un bug a été signalé par le jury. Ce problème survenait lors des déplacements du personnage. En effet, étant donné que le jeu se déroule en 2D, il était possible pour n'importe quel joueur de traverser les arbres, les bâtiments et les objets. Cela posait un

réel problème de satisfaction, car les joueurs ne percevaient pas de limites virtuelles dans le jeu. Les barrières et les maisons perdaient donc leur intérêt dans le bon déroulement de la partie.

Ce problème provenait du fait que tous les graphismes présents sur la carte étaient de simples images 2D, sans aucune dimension physique réelle associée.

La première solution à laquelle nous avons pensé a été de nous renseigner sur des sites spécialisés et des forums, où d'autres développeurs avaient déjà rencontré ce type de difficulté. Nous avons ainsi découvert le composant BoxCollider intégré au moteur Unity.

Pour l'utiliser, il faut définir la zone de collision pour chaque objet. Ainsi, dès qu'un personnage se déplace, le moteur gère automatiquement les calculs nécessaires pour détecter les collisions entre objets. Il est donc indispensable de délimiter précisément la zone de collision pour chaque élément.

Cependant, contrairement aux exemples classiques trouvés en ligne, nos personnages n'avaient pas de représentation fixe, car ils sont clonés dynamiquement en raison de l'implémentation de notre système multijoueur. Après plusieurs essais, nous avons réussi à intégrer ce système natif du moteur, mais un problème subsistait : lorsqu'un joueur se rapprochait d'un obstacle, il ne pouvait pas le traverser, mais dans certaines situations, un bug survenait et le personnage se retrouvait téléporté à travers l'obstacle.

Ne répondant pas à nos attentes, nous avons finalement décidé d'abandonner cette solution et de développer notre propre système de gestion du franchissement des obstacles.

Pour résoudre ce problème, qui impactait négativement l'expérience utilisateur, nous avons décidé d'étudier la question plus en profondeur.

La deuxième solution envisagée a été de créer une classe Obstacle possédant des attributs représentant ses coordonnées en haut à gauche et en bas à droite. Cela permettrait de définir un rectangle correspondant à la zone occupée par l'obstacle sur la carte (maison, arbre, objet, etc.).

Ainsi, au lancement de la partie, nous instancions tous ces rectangles aux emplacements des obstacles présents. Ensuite, lorsqu'un joueur souhaite se déplacer, nous récupérons les nouvelles coordonnées (x, y) de la position visée. Si ce point se trouve à l'intérieur de l'un des rectangles obstacles, le déplacement est interdit.

Pour vérifier cela, il suffit de parcourir la liste des obstacles et de vérifier si la coordonnée x du joueur se situe entre les bornes en x du rectangle, et si la coordonnée y est comprise entre les bornes en y du rectangle. Cette méthode est simple à implémenter et à comprendre.

Cependant, elle présente plusieurs inconvénients majeurs :

Le calcul doit être réalisé très fréquemment (à chaque déplacement) et

pour chaque obstacle, ce qui peut rapidement devenir coûteux en ressources, impactant les performances du client. Cette approche nécessite de référencer explicitement toutes les zones infranchissables dans des rectangles distincts. Si la carte venait à changer, il faudrait modifier non seulement les éléments graphiques, mais aussi ces rectangles invisibles, rendant la maintenance fastidieuse. Pour ces raisons, malgré sa simplicité, nous avons décidé d'abandonner cette méthode au profit d'une autre solution plus performante et évolutive.

Une seconde idée nous est alors apparue : réaliser une matrice de valeurs booléennes représentant chaque case de la carte, afin de savoir si une case est empruntable par un joueur ou non. Seules les cases de type Road — correspondant aux chemins de terre visibles sur la carte — sont traversables. Ces chemins représentent plus de 300 cases, soit environ 10 % de la surface du premier niveau.

Le principal avantage de cette méthode est qu'elle évite de redéfinir explicitement toutes les zones infranchissables. En effet, les objets et terrains sont déjà organisés dans des dossiers bien nommés, ce qui nous a permis d'identifier leur type facilement. Lors du chargement de la carte, une fonction statique est exécutée une seule fois pour remplir cette matrice. Cette fonction parcourt toutes les cases de la carte et marque comme franchissables uniquement celles correspondant aux critères définis.

En C, une matrice de booléens est, par défaut, stockée avec un octet (8 bits) par case, ce qui représente un certain gaspillage de mémoire. De plus, comme certaines coordonnées de la carte sont négatives, nous avons dû recentrer la matrice pour que l'élément d'index (0,0) corresponde bien à la case située aux coordonnées (0,0) dans l'espace du jeu. Cela nécessite donc de connaître la largeur et la hauteur de la carte afin de créer une marge initiale.

Pour optimiser davantage la mémoire, nous avons décidé d'implémenter une matrice de bits, où chaque case est codée par un seul bit (0 ou 1), suffisant pour notre usage.

Lors du déplacement d'un joueur (dans la classe PlayerMovement), lorsque celui-ci appuie sur une touche de déplacement, ses nouvelles coordonnées théoriques sont calculées en flottants, puis arrondies à l'entier inférieur. Cette simplification facilite les calculs. Une fonction statique est ensuite appelée pour vérifier si, à ces coordonnées, un obstacle est présent (en consultant la matrice de bits).

Cette méthode, grâce à la matrice pré-calculée, est très rapide et permet au joueur de se déplacer de façon fluide tout en respectant les obstacles.

Ainsi, dans la version finale de notre jeu, les joueurs ne peuvent plus traverser les maisons, arbres et objets. Ce changement rend le gameplay plus dynamique et réaliste, introduisant des contraintes qui augmentent la

difficulté. Ce point, qui avait été soulevé lors de la soutenance précédente, a donc été corrigé, améliorant significativement la satisfaction des joueurs.

4.5 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. Lorsqu'un joueur termine la partie, tous les joueurs connectés sont redirigés vers la page de fin, les informant que le niveau est terminé. Ils sont alors invités à cliquer sur le bouton "Retour au menu" qui les redirige vers le menu principal de notre jeu. Ainsi ils peuvent refaire une nouvelle partie. Une capture d'écran de la page des menus est disponible en annexe, figure 8.

4.6 Graphismes

Les graphismes jouent un rôle essentiel dans un jeu, car ils contribuent à offrir une expérience fluide et immersive au joueur. Pour cela, nous avons décidé de représenter notre personnage par une animation plutôt que par une simple image statique. Celui-ci est désormais composé d'une succession de sept images qui, répétées en boucle, créent un effet d'animation continue.

Par ailleurs, nous avons enrichi notre carte en y ajoutant divers éléments de décor. Pour cela, nous avons recherché des images adaptées à l'ambiance de notre jeu et les avons intégrées dans les différents composants de notre TileMap, un outil graphique nous permettant de dessiner efficacement nos niveaux.

Ces éléments graphiques sont particulièrement importants, car ce sont souvent les premiers détails observés par le joueur. Ils influencent son impression initiale sur le jeu, rendant le soin apporté à ces aspects primordial.

Tout au long du développement, nous avons cherché à améliorer continuellement l'expérience utilisateur, notamment en perfectionnant les graphismes. Pour cela, nous avons choisi d'utiliser une bibliothèque d'assets libres de droits, trouvée en ligne, afin de créer un univers graphique inspiré des contes des frères Grimm. Cette collection d'assets médiévaux a été importée dans Unity, version 6, notre moteur de jeu.

4.7 Les TileMaps

Souhaitant réaliser un jeu en vue 2D, nous avons choisi d'utiliser les Tilemaps, un outil natif proposé par Unity. Une Tilemap est constituée d'une grille composée de tuiles carrées de taille uniforme. Cela nous permet de construire facilement des environnements cohérents et modulaires.

Nous avons ainsi créé différentes palettes de tuiles, chacune correspondant à une ambiance particulière. Au total, nous avons réalisé quatre palettes distinctes afin de varier les décors et enrichir visuellement nos niveaux.

- Les chemins : Les éléments de cette palette correspondent à l'ensemble des chemins empruntables par les joueurs et les loups. Généralement constitués de terre, ces chemins se distinguent par une teinte légèrement marron, ce qui les rend facilement identifiables sur la carte.
- Les herbes : Cette palette regroupe l'ensemble des tuiles représentant l'herbe et la forêt. Ces zones ne sont pas empruntables par les joueurs.
- Les arbres et les plantes : De nombreuses plantations sont visibles dans le jeu. Elles sont toutes répertoriées dans cette palette. Généralement disposées sur des zones d'herbe, elles ne sont pas empruntables par les joueurs.
- Les Décors : De nombreux objets sont présents sur la carte (panneaux, lumières, bancs, boîtes, feux de camp, etc.). Représentant de véritables éléments du décor, ils ne peuvent pas être traversés par les joueurs.
- La mer : À certains endroits de la carte, la mer est légèrement visible. Évidemment, il est impossible pour les joueurs de la traverser à pied.

Toutes les tuiles non empruntables pour les joueurs le sont également pour les animaux. Ainsi, un loup qui suit un joueur empruntera uniquement les chemins autorisés, en utilisant l'algorithme de Dijkstra, comme présenté dans la partie Intelligence artificielle .

4.8 Interface

Lorsque le joueur ouvre notre jeu, il est immédiatement dirigé vers la page du menu principal, qui occupe toute la taille de la fenêtre.

Sur cette page, trois boutons sont présents ainsi qu'un champ de texte. Un bouton permettant de quitter l'application se trouve en haut à gauche. Au centre, un champ permet de saisir le token d'une partie existante, accompagné d'un bouton pour rejoindre cette partie une fois le code entré. Enfin, un dernier bouton situé en bas de l'écran permet de créer une nouvelle partie.

Une fois en jeu, toute la fenêtre est entièrement dédiée à l'environnement de jeu. Le joueur y découvre une vaste carte boisée affichée en plein écran. En haut de cette interface, une fine barre légèrement grisée présente deux informations essentielles.

À gauche de cette barre figurent les points de vie du joueur, dont la valeur peut varier de 0 à 100. Par exemple, si vous possédez 90 points de vie et utilisez une potion censée en restaurer 50, votre total n'atteindra que 100, la valeur maximale autorisée. La potion est consommée, mais les points excédentaires sont perdus.

À droite, on retrouve le token de la session en cours. Ce code unique peut être partagé avec vos amis. En le saisissant dans le champ prévu du menu principal, ils peuvent rejoindre directement votre partie. Cela permet de jouer facilement en multijoueur et de vivre l'aventure à plusieurs.

4.9 Rendu visuel du produit

Pour avoir une certaine cohérence graphique entre nos différents produits, nous avons fait le choix de développer une charte graphique. Pour ce faire, nous avons fait une sélection de couleurs, de types d'images et de polices d'écriture. Tous ces éléments sont présents dans notre jeu, sur notre site web, mais également dans nos logos.

Avec une volonté de créer une charte graphique autour des contes des frères Grimm, nous avons dans un premier temps recherché comment les gens s'imaginent ces contes. Nous avons regardé des livres pour enfants et observé leurs illustrations. De nombreuses couleurs sombres en ressortaient.

Afin de rendre le jeu plus attrayant pour un jeune public, nous avons fait le choix d'utiliser des couleurs plus dynamiques et colorées : du rouge, du vert, du beige ! Toutes ces couleurs ont rendu notre charte graphique plus vivante. Pour la couleur de fond de nos logos et de l'écran de chargement du jeu, nous avons choisi un beige clair, discret mais chaleureux.

Pour notre jeu, nous avons donc trouvé une palette de composants médiévaux qui partageait le même esprit que celui que nous voulions exprimer. Nous avons alors sélectionné cette palette libre de droits.

Pour notre site web, nous nous sommes concentrés sur l'accessibilité de notre produit. Nous avons voulu y placer de nombreuses images afin d'illustrer au maximum l'univers de notre jeu, dans l'objectif de donner envie aux visiteurs de le télécharger. De nombreuses couleurs sont également présentes sur le site pour le rendre plus attrayant. Nous avons tout de même fait le choix de rester élégants et fonctionnels.

Pour nos logos, nous avons également appliqué notre charte graphique. Tout d'abord, notre logo de groupe "The Grims", en noir et blanc, représente

notre équipe et notre studio de création. Avec notre nom au centre, nous avons ajouté une tête de loup à gauche pour faire un clin d'œil à notre projet. En effet, passionnés par les contes, notre équipe s'est développée autour de cette thématique.

Le logo en noir et blanc est assez sombre de prime abord, mais il permet de mettre en valeur le logo de notre projet. Le logo de "Once Upon a Time" est donc né pendant le développement de notre jeu. Représentant le jeu lui-même et non pas notre équipe, il reprend toutes les caractéristiques graphiques de notre projet. On y voit au centre le nom du jeu, à gauche une tête de loup faisant référence au Petit Chaperon Rouge, et l'ensemble est entouré de nombreux arbres et feuillages. Ce second logo fait un rappel visuel fort sur le principe fondamental de notre jeu : les contes de Grimm et leurs univers mythiques.

Grâce à notre recherche graphique, nous sommes parvenus à créer une ambiance cohérente et reconnaissable sur tous les supports liés à notre jeu, ce qui le rend encore plus attrayant.

4.10 Composants présents dans le jeu

Afin d'avoir des interactions dans le jeu, de nombreux items sont présents, catégorisés en trois grandes familles. Chaque composant se distingue du décor par le fait qu'il peut interagir avec les joueurs.

Tout d'abord, nous avons les "bâtiments", qui regroupent les feux de camp et les coffres.

- Le feux de camp : Animé par une flamme rouge et entouré de pierres, le feu de camp permet de rendre de la vie aux joueurs qui s'en approchent. Tant qu'un joueur reste à proximité, il récupère progressivement des points de vie.

- Le coffre : Représenté par une petite boîte en bois, le coffre peut contenir de nombreux items. Lorsqu'un joueur s'en approche, il récupère automatiquement le premier objet présent à l'intérieur.

  Chaque joueur ne pouvant tenir qu'un seul objet dans sa main droite, s'il s'approche d'un coffre avec un objet déjà en main, il le laissera tomber pour prendre celui contenu dans le coffre.

Le second type de composants correspond aux créatures. Ce sont des êtres vivants capables de se déplacer sur la carte.

- Le loup : Le loup est le premier ennemi des joueurs dans notre jeu. En effet, lorsque l'on s'approche d'un loup, celui-ci vous attaque et vous

inflige des dégâts. S'il vous poursuit, il ne vous lâchera pas tant que vous aurez des points de vie.

Lorsque vous êtes proche d'un loup, vous pouvez le frapper en appuyant sur la touche "F". Chaque coup inflige 5 points de dégâts.

- La Personne : Les Personnes sont les joueurs qui jouent en ligne avec vous. Vous pouvez les voir bouger en temps réel, mais vous ne pouvez pas interagir directement avec eux.

Le dernier type de composants correspond aux items. Ces objets peuvent se trouver par terre dans le jeu, dans un coffre, ou simplement dans la main d'un joueur.

Lorsque vous tenez un item en main, vous pouvez appuyer sur la touche "P" pour le jeter par terre. Ainsi, un autre joueur pourra le ramasser et l'utiliser.

- L'épée : L'épée est une arme que les joueurs peuvent utiliser pour infliger davantage de dégâts aux ennemis. Avec une épée, un loup reçoit 40 points de dégâts. Après 2 utilisations, l'épée se casse et ne peut plus être utilisée.

- La potion de soin : Représentée par une petite fiole bleue, la potion de soin permet de régénérer instantanément 50 points de vie à celui qui la consomme. Après utilisation, elle disparaît, car il ne reste plus de potion.

- La galette : Cette galette est la clé du conte. Lorsque vous l'avez en main, rendez-vous à la maison de la grand-mère pour terminer le conte. Tant que la grand-mère n'a pas reçu sa galette, le jeu continue.

4.11 Interaction avec les objets

Afin de pouvoir interagir avec les composants présents dans le jeu, nous avons créé un système dédié.

Pour référencer la manière dont les joueurs peuvent interagir avec ces composants, nous avons défini une énumération des différents types d'interaction. Les deux principales façons d'interagir sont WhileNearby et OneTime.

WhileNearby correspond au fait d'interagir avec un composant tant que le joueur se trouve à proximité de celui-ci. OneTime permet d'interagir avec un composant une seule fois lorsqu'on s'en approche. Évidemment, si l'on s'éloigne puis revient à proximité, il est possible d'interagir à nouveau. Pour

ces deux méthodes, il est nécessaire de se trouver à moins d'une tuile du composant concerné pour pouvoir interagir avec lui.

Ces informations concernant la méthode d'interaction sont enregistrées dans un attribut dédié des classes héritant de PositionedItem. Cela concerne, par exemple, l'épée, la potion ou la galette.

La méthode WhileNearby est la plus simple à développer. Dans la classe Room, la fonction Update() est appelée à chaque frame. Cela nous permet de vérifier en permanence, pour tous les composants dont l'interaction se fait de manière continue, si le joueur actuel est proche de l'un de ces objets.

Si c'est le cas, le joueur va interagir automatiquement avec cet objet, en utilisant sa fonction Interact(). Cette fonction réalise une succession de tâches spécifiques à chaque objet. Par exemple, le feu de camp génère des points de vie, tandis que le loup inflige des coups.

Ainsi, nous effectuons une interaction en permanence tant que le joueur reste à proximité.

Pour les composants dont l'interaction se fait en "OneTime", cela signifie que le joueur peut interagir une seule fois lorsqu'il s'approche de celui-ci. Il pourra interagir à nouveau uniquement s'il s'éloigne d'au moins une case, puis revient.

Pour gérer et contrôler le nombre d'interactions, nous nous basons sur la fonction de mouvement du joueur. Ainsi, lorsqu'un joueur se déplace, il regarde autour de ses nouvelles coordonnées s'il y a des composants de type "OneTime" dont l'attribut HasTriggered est à faux. Cela signifie que le joueur peut entrer en interaction avec le composant, et l'attribut HasTriggered passe alors à vrai.

Si le joueur reste à côté du composant, HasTriggered reste à vrai.

Lorsque le joueur se déplace à nouveau, il vérifie pour chaque composant "OneTime" s'il se situe toujours à côté. Si ce n'est plus le cas, il remet l'attribut HasTriggered à faux, ce qui permettra au joueur de réinteragir avec cet objet lors d'un prochain passage.

Avec ces deux méthodes, nous avons pu mettre en place différentes techniques de jeu pour les joueurs, ce qui rend Once Upon a Time plus attractif et dynamique.

4.12 Optimisation du jeu

4.12.1 Les préfabriqués

Au fur et à mesure du développement de notre jeu, nous nous efforçons de le rendre plus rapide et moins gourmand en énergie. Pour ce faire, nous

avons réalisé de nombreuses optimisations conséquentes depuis les soutenances précédentes.

Tous les personnages et objets (coffres, épées, galette...) sont des préfabriqués. Pour chaque objet, nous avons une animation du personnage le tenant dans sa main.

Dans les versions précédentes de notre jeu, à chaque réception d'un nouveau message concernant un changement de position d'un joueur ou simplement des informations le concernant, nous le détruisions puis le clonions à nouveau. Cette méthode, certes facile à mettre en place, posait de nombreux problèmes. En effet, si nous voulions uniquement mettre à jour ses coordonnées, nous effectuions de lourds calculs pour un résultat minime.

Désormais, lorsqu'un joueur envoie de nouvelles données, nous vérifions si son apparence doit changer. Si c'est le cas, nous instancions un nouveau préfabriqué. Dans le cas contraire (près de 95 % des cas), nous nous contentons de mettre à jour ses positions et d'utiliser nos algorithmes de déplacement fluide afin de rendre le tout plus agréable.

4.12.2 Les requêtes

Essentiellement basé sur le multijoueur et le jeu en équipe, notre jeu a fondamentalement besoin d'effectuer des requêtes pour fonctionner. Cependant, il est nécessaire de faire un tri, car tous les changements ne justifient pas d'en informer tous les autres joueurs.

Prenons l'exemple des déplacements : si un joueur reste appuyé sur une flèche pour se déplacer, l'algorithme de déplacement sera exécuté à chaque frame. Pour éviter un surplus de requêtes, nous avons choisi de mettre en place une limite.

Sachant qu'environ 500 frames peuvent être réalisées par seconde, il est essentiel de filtrer ces données. Nous avons donc placé un compteur initialisé à 0. Dès qu'un mouvement est réalisé, nous l'incrémentons. Si ce compteur modulo 100 est égal à 0, nous laissons passer la requête et envoyons ainsi ces informations à tous les joueurs.

Cette simple technique nous permet de réduire considérablement le nombre de requêtes à l'API et d'améliorer les performances.

4.12.3 Intelligence artificielle

Afin d'optimiser notre système d'intelligence artificielle, nous limitons l'exécution des algorithmes intelligents. En effet, à chaque frame du jeu, les réactions du loup sont calculées, ce qui permet de déterminer l'action qu'il va réaliser. Pour rappel, chaque loup suit un participant particulier, ou personne.

Pour limiter les calculs, seul le participant concerné par l'attaque calcule son itinéraire et le partage ensuite avec les autres joueurs. Cela permet de gagner en efficacité et d'éviter des bugs de répétitions infinies lors des déplacements d'un loup.

5 Site web

5.1 Introduction du site web

Afin de répondre aux exigences du cahier des charges, nous avons développé un site web fonctionnel, accessible à l'adresse suivante : the-grimms.com.

5.2 Présentation du site web

Le site web est composé de quatre pages.

La première page est celle par défaut. Lors de son ouverture, on voit au premier plan le nom de notre jeu, accompagné d'un bouton Télécharger qui redirige vers la page des téléchargements. En arrière-plan, une image extraite de notre jeu est affichée.

En faisant défiler la page vers le bas, on découvre différentes images du jeu accompagnées de textes descriptifs, visant à donner envie aux visiteurs d'essayer notre jeu. De nombreuses couleurs vives rappellent l'ambiance générale de notre projet Once Upon a Time .

La deuxième page met en avant les membres du groupe, avec une photo de profil, une brève description et les tâches associées à chacun d'eux. Chaque membre est accompagné d'une créature ou d'un objet présent dans le jeu.

La troisième page présente l'avancement de notre projet ainsi que les événements importants passés. Avec près d'une dizaine d'étapes clés, le visiteur peut découvrir l'histoire derrière notre projet.

Enfin, la dernière page permet de télécharger la dernière version du jeu, le rapport de soutenance ainsi que les manuels d'installation et d'utilisation du jeu.

5.3 Développement du site web

L'ensemble du site web a été développé en PHP 8.4. Une base de données y a été créée afin d'ajouter des données de manière dynamique. Par exemple, 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 pour chacun.

Cette méthode est également utilisée pour enregistrer les différents éléments de la timeline, c'est-à-dire la liste des événements passés jusqu'à aujourd'hui.

Développer un site web peut être une étape fastidieuse et répétitive. De nombreuses failles de sécurité peuvent apparaître rapidement et sont souvent longues à corriger. C'est pour ces raisons que nous avons choisi d'utiliser le framework PHP Symfony 7.

L'utilisation des frameworks s'est largement démocratisée ces dernières années. En effet, ils permettent d'intégrer des modules et composants développés par une communauté active, souvent bénévole.

Le choix de Symfony plutôt qu'un autre framework s'est fait en raison des nombreuses ressources gratuites disponibles en ligne. Presque tous les problèmes rencontrés ont déjà une solution sur des forums ou sites d'apprentissage. De plus, sa documentation bien développée nous permet de progresser rapidement dans la conception du site.

Symfony 7 est un framework dit ORM (Object-Relational Mapping). Ce mode de fonctionnement crée une correspondance entre un objet et un modèle relationnel de base de données. De nombreuses méthodes simplifient la réalisation de requêtes SQL.

Le framework suit une architecture précise, 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, notamment pour la gestion des tables de la base de données et les Controllers, qui contiennent la logique du site.

Pour tout le rendu côté client, c'est-à-dire le HTML et la mise en forme graphique, l'ensemble des ressources se trouve dans le dossier Template. De nombreux autres dossiers et fichiers servent à la configuration et à la gestion des divers éléments du site.

Après un premier développement en local sur nos ordinateurs, nous avons choisi de publier le site sur internet. Pour cela, nous avons décidé de l'héberger sur un serveur auquel nous avions déjà accès avant le début du projet.

Après avoir configuré notre site avec les informations du serveur, nous disposons désormais d'une base de données en production reliée à un site web pleinement fonctionnel.

5.4 Différence avec notre API

Afin de bien distinguer la partie API fonctionnelle du système multijoueur de notre jeu et notre site web de présentation du projet, nous avons choisi de séparer ces deux services sur deux domaines différents.

Le premier est accessible via le sous-domaine api.the-grimms.com, tandis que notre site d'équipe se trouve à l'adresse the-grimms.com.

Cette séparation facilite la visualisation des parties sur lesquelles nous travaillons, tout en évitant des problèmes de fonctionnement. En effet, en restant discrets pour les joueurs, le domaine api.the-grimms.com ne traite que les données multijoueurs, tandis que the-grimms.com sert de vitrine accessible à tous.

Les deux sites sont donc totalement dissociés et indépendants l'un de l'autre.

5.5 Conclusion site web

Avec notre site web accessible en ligne, tout le monde peut découvrir l'équipe The Grimms à travers son histoire et ses descriptions captivantes. Grâce à la possibilité de télécharger notre jeu Once Upon a Time, notre site devient une véritable porte d'entrée pour tous les joueurs souhaitant redécouvrir les contes des frères Grimm autrement.

6 Procédure d'installation

Comme exigé dans le cahier des charges, nous avons réalisé une procédure d'installation automatique de notre jeu.

L'objectif de ce système est de proposer un simple fichier exécutable permettant d'installer proprement la dernière version de notre jeu sur la machine du client, après téléchargement depuis notre site web.

Pour réaliser cet installateur, nous avons choisi d'utiliser le logiciel libre Inno Setup. Facile à prendre en main, il nous a permis de créer rapidement notre exécutable. Pour cela, nous avons renseigné les informations concernant notre jeu ainsi que les fichiers sources correspondants. Nous avons aussi écrit l'ensemble des messages affichés lors des différentes étapes de l'installation.

Nous avons fait le choix de proposer une installation bilingue, en anglais et en français, afin de permettre à l'utilisateur de choisir sa langue.

Si l'utilisateur exécute une nouvelle fois cet installateur, il peut choisir soit de réinstaller la dernière version du jeu, soit de le supprimer proprement de son ordinateur.

Une fois les caractéristiques définies, nous avons compilé notre exécutable avec le logiciel prévu à cet effet.

Après avoir obtenu ce fichier, nous l'avons mis à disposition sur notre site, permettant ainsi le téléchargement du jeu à l'adresse suivante : thegrimms.com/download.

Un guide complet d'utilisation de la procédure d'installation est également disponible sur cette même page.

7 Les problèmes rencontrés

7.1 Multijoueur (sérialisation, désérialisation)

L'un des problèmes qui nous a pris le plus de temps a été la sérialisation et la désérialisation des informations. En effet, ne maîtrisant pas ces méthodes en C, nous n'envoyions au départ que les valeurs x et y, ce qui ne posait aucun problème. Cependant, dès que nous avons voulu synchroniser une partie entière, il a fallu trouver une solution plus robuste.

Nous avons alors découvert le principe de sérialisation et de désérialisation intégré directement dans C. Ce mécanisme permet de convertir rapidement et efficacement des objets en JSON, ce qui facilite grandement l'échange de données complexes entre clients et serveur.

Après avoir été bloqués pendant de nombreuses heures sur ce sujet, nous avons finalement réussi à mettre en place cette méthode, ce qui nous a permis d'améliorer considérablement la gestion des données dans notre jeu.

7.2 Désérialisation des objets

Avec une structure de jeu complexe, de nombreuses classes possèdent des attributs dont les types sont eux-mêmes des classes. Lorsque nous développons en C, cela ne pose aucun problème car ce langage est orienté objet, et avec l'utilisation de Rider, nous pouvons naviguer facilement d'une classe à une autre en utilisant le point . entre un objet et un autre.

Cependant, lorsque nous avons développé la classe Person, représentant un joueur, nous lui avons ajouté un attribut RightHandItem de type Item. Grâce à cet attribut, le joueur peut porter un objet dans sa main.

Lorsque nous sérialisons l'objet Person, il est correctement transformé en JSON, avec un sous-objet RightHandItem contenant les attributs de cet objet. L'envoi et la réception du message fonctionnent parfaitement, cependant, la désérialisation ne marchait pas.

En effet, Item étant une classe parent, le programme ne savait pas quel objet enfant devait être instancié lors de la désérialisation. Pour résoudre ce problème, nous avons développé un convertisseur appelé ItemConverter, qui permet de convertir un Item en son objet enfant réel.

Pour différencier les enfants entre eux (comme Sword, LifePotion, Cake, etc.), nous avons ajouté un attribut textuel nommé Type, reprenant le nom sous forme de chaîne de caractères de l'objet. Ainsi, lors du passage par le convertisseur, celui-ci regarde la valeur de Type et retourne une nouvelle instance de l'objet correspondant avec les données adéquates.

Comprendre ce principe de désérialisation d'objets imbriqués a été une

étape fastidieuse, car c'était une notion encore inconnue pour nous à ce moment du développement.

La désérialisation des données a été une étape complexe dans le développement de notre jeu. En effet, avec toutes ces contraintes et règles particulières, nous avons dû prendre du temps afin de tout faire correctement.

Une contrainte de ce système nous a fait perdre beaucoup de temps. La classe Item, étant abstraite, ne peut pas être instanciée directement, car cela n'a aucun sens de créer un objet dans le jeu sans lui définir concrètement des caractéristiques. C'est pourquoi nous avons fait le choix de créer des classes enfants héritant de cette classe Item, afin de pouvoir définir ces caractéristiques correctement.

Cependant, lors de la sérialisation des données de la classe Person, l'attribut RightHandItem doit également être désérialisé. Cela crée donc un nouvel Item, tout cela en passant par le convertisseur approprié. Cependant, sachant qu'un objet abstrait ne peut pas être instancié directement, notre processus ne fonctionnait pas. Cela générait une erreur dans le processus et affichait un message dans la console de l'éditeur de jeu.

Pour pallier ce problème, deux solutions étaient envisageables : la première consistait simplement à transformer la classe abstraite Item en une classe simple, et la seconde était de modifier le processus de transformation des objets dans le convertisseur.

Pour des raisons de clarté et de logique, nous avons décidé d'appliquer la seconde solution, ce qui nous a pris un peu plus de temps. Après plusieurs heures à chercher la source du problème, nous sommes parvenus à le résoudre en analysant intégralement, du début à la fin, l'ensemble des actions réalisées.

Désormais, toute la sérialisation se déroule comme prévu, et aucun problème n'est à signaler.

7.3 Importation de librairies à jour

Au début du projet, lorsque nous avons décidé de la méthode à utiliser pour notre système multijoueur, nous avons commencé sa mise en place. Pour rappel, notre choix s'était porté sur PusherJS pour plusieurs raisons expliquées dans la section Système de Multijoueur .

En parcourant la documentation de PusherJS, nous avons découvert avec surprise qu'une version spécifique pour Unity était proposée, ainsi qu'une bibliothèque complète open source disponible sur GitHub. Enthousiastes, nous avons donc décidé d'utiliser ce code pour gagner du temps.

Après une rapide importation de la librairie dans l'éditeur Unity, nous avons pu utiliser Pusher très simplement. Une fois notre API configurée et notre système de messages en trois étapes en place, nous sommes parvenus à

envoyer et recevoir des messages en temps réel grâce au système d'émulation intégré à Unity.

Cependant, après plusieurs heures de développement sur d'autres composants, nous avons tenté pour la première fois de compiler notre jeu. Le processus s'est interrompu brutalement, affichant plusieurs messages d'erreur liés aux bibliothèques importées pour la communication via Pusher.

À ce stade du développement, après des dizaines d'heures passées sur ce modèle, notre inquiétude est montée. En effet, avec le système actuel, il était impossible de compiler notre jeu. Nous devions réagir rapidement.

Après de nombreuses tentatives d'importer des versions plus récentes, de modifier manuellement des fichiers sources, et même d'entrer en contact avec le service de mise à jour des librairies chez Pusher, rien n'a fonctionné. Nous n'avons pas réussi à résoudre le problème.

Déterminés à faire fonctionner le système multijoueur, nous avons envisagé une autre approche. Sachant que la bibliothèque Pusher était dépréciée, nous avons étudié comment la remplacer. Parallèlement, nous nous sommes informés sur le fonctionnement des WebSockets.

Face à la complexité du maillage imposé par Pusher, nous avons décidé de conserver la même architecture de base mais de développer notre propre serveur Node.js. En trouvant un exemple simple sur Internet, nous avons rapidement mis en place un serveur Node.js en local.

De notre côté, en C, avec l'exemple et les librairies WebSocket associées, nous avons pu envoyer des messages. Par un heureux hasard, en combinant les librairies WebSocket de l'exemple et celle de PusherJS, la compilation avec tous les messages PusherJS fonctionnait enfin.

Nous avons alors compris que les deux librairies tentaient d'importer la même dépendance, mais que la plus récente, installée en premier, était prise en compte et la seconde ignorée, ce qui a permis de contourner le problème.

C'est ainsi, après avoir testé différentes méthodes et techniques, que nous avons finalement résolu ce blocage de compilation.

Depuis cet épisode, survenu début 2025, nous n'avons plus rencontré de problème de compilation. Nous avons donc pu poursuivre le développement, laissant derrière nous cette étape riche en émotions.

7.4 Utilisation de l'outil Git

La collaboration entre les différents membres du groupe s'est avérée difficile à gérer. Après avoir réfléchi ensemble aux idées du jeu, il était essentiel de trouver un moyen efficace pour avancer simultanément sur le projet.

Pour cela, nous avons utilisé un repository GitLab mis à notre disposition. Cependant, la prise en main de cet outil s'est révélée complexe. En

effet, si tous les membres envoyaient leur code directement sur la branche principale, de nombreux conflits apparaissaient lorsqu'ils travaillaient sur les mêmes fichiers.

Pour pallier ce problème, nous avons exploité le système de branches proposé par Git. Nous avons structuré notre repository ainsi :

La branche master, qui contient une version finale du projet destinée à chaque soutenance, La branche dev, qui regroupe l'ensemble des avancées en cours. Lorsqu'un membre souhaite ajouter une fonctionnalité, il crée une branche spécifique, par exemple feature-x , issue de la dernière version de la branche dev. Une fois ses modifications terminées, il fusionne (merge) cette branche dans dev, qui devient alors la dernière version de développement.

Lorsque toutes les fonctionnalités prévues sont intégrées, nous réalisons un unique push de la branche dev vers master, garantissant ainsi une version stable pour la soutenance.

Cette méthode, bien que très efficace, demande de la pratique pour être bien maîtrisée. À plusieurs reprises, nous avons perdu des fichiers sur lesquels nous avions travaillé. Heureusement, en consultant les logs du repository, il est possible de retrouver les versions précédentes et de restaurer les travaux perdus.

Ainsi, tant que nous effectuons régulièrement des commits, nous pouvons récupérer tout travail supprimé accidentellement.

7.5 Coordination du groupe

Durant la phase de réflexion et de développement du projet, nous avons rencontré certaines difficultés liées à la coordination entre les membres du groupe. En effet, après le départ d'Anir PERROD, notre équipe s'est retrouvée réduite à quatre personnes pour accomplir l'ensemble des tâches requises. De plus, comme la plupart d'entre nous ne maîtrisaient pas encore bien les outils comme Git, il a été difficile de travailler efficacement tous ensemble sur le projet.

8 Récit de la réalisation

8.1 Introduction

Avec une idée initiale appréciée par tous les membres du groupe et une volonté commune de développer un projet soigné, chacun a été conquis dès le départ. Cette idée est née d'une discussion lors d'un temps dédié en cours de méthodologie.

8.2 Membres du groupes

Avec des membres aux qualités variées, notre groupe disposait dès le départ des compétences nécessaires pour réaliser un projet complet.

Anir Perrod, attiré par le côté artistique, nous a apporté ses talents de dessinateur, ce qui nous a permis de créer notre premier logo : le fameux logo "The Grimms" en noir, avec une tête de loup sur le côté.

Mantas Krukonis, passionné par les jeux vidéo et les histoires, a élargi notre champ de vision en imaginant des scénarios plus créatifs tout au long de la période de réflexion et de développement.

Lucas Morel, séduit par l'aspect technique du projet, était enthousiaste à l'idée de travailler sur un projet mêlant technicité et création.

Nicolas Simeonov, avec son regard extérieur, a permis de faire émerger de nouvelles réflexions autour de notre jeu.

Enfin, Alexandre Ledoux, grâce à son appétence pour le développement, a été un pilier dans la réalisation concrète du jeu.

8.3 Répartition des tâches

Avec un planning correctement réparti entre les membres du groupe, les tâches individuelles restaient accessibles à tous. De plus, grâce à nos discussions régulières lors des temps dédiés au développement du projet pendant les cours de méthodologie, nous étions tous d'accord pour nous entraider en cas de difficulté.

Au début de l'année, vers octobre 2024, nous avons établi cette répartition des tâches entre les membres du groupe.

Cette répartition des tâches, validée par M. Christophe Boullay, a constitué la base de notre développement du projet. La première colonne, à gauche, regroupe les différentes tâches représentant le développement global du jeu vidéo. Chaque colonne suivante correspond aux tâches individuelles attribuées aux membres du groupe.

Chaque membre peut être responsable ou suppléant d'une tâche, ou ne pas être assigné à une tâche spécifique.

Pour assurer une répartition équitable et harmonieuse entre tous les membres, nous avons décidé que chacun serait responsable d'au moins deux tâches et suppléant sur trois autres. Ainsi, chaque membre travaille sur cinq parties différentes du projet.

8.4 Réorganisation du groupe

Après le départ soudain d'Anir Perrod, qui était particulièrement impliqué dans le développement des graphismes et des éléments artistiques, nous avons dû réorganiser notre travail. Soucieux de respecter les délais fixés en début d'année dans notre planning de projet (voir page 5), nous avons décidé d'accélérer la réalisation des graphismes.

Après notre première soutenance technique, alors que nous n'étions plus que quatre, nous avons constaté un écart important entre le planning initial et l'avancement réel du développement. Pour pallier cette situation, Alexandre Ledoux a pris en charge le développement complet du système multijoueur afin de fournir une base de code fonctionnelle.

8.5 Tâches supplémentaires

D'autres tâches, non présentes sur ce tableau, sont également à prendre en compte. En effet, la recherche du scénario du jeu, le développement du système d'installation et de désinstallation automatique, ainsi que plusieurs autres éléments, n'étaient pas encore connus au moment de l'élaboration de ce tableau. De plus, certains composants ont pris plus de temps à être réalisés que prévu. Nous pensons notamment au développement du système multijoueur, mais aussi au gameplay.

9 Axe d'amélioration technique du jeu

Après avoir finalisé notre jeu pour la dernière soutenance technique, nous avons encore de nombreuses idées pour le rendre plus fluide et réactif. En effet, un projet de cette envergure n'est jamais vraiment terminé et nécessite des mises à jour régulières afin d'en assurer le bon fonctionnement.

9.1 Optimisation de l'API

Afin de rendre notre jeu encore plus rapide et réactif en multijoueur, plusieurs méthodes d'optimisation sont envisageables. Actuellement, notre architecture (voir figure 5) repose sur des requêtes TCP envoyées vers notre API, qui elle-même communique avec l'API de Pusher.

Dans un premier temps, pour gagner en rapidité, l'utilisation du protocole UDP serait plus intéressante. En effet, ce protocole, en n'assurant aucune protection contre la perte de paquets en chemin, présente l'avantage de livrer les données plus rapidement. Cela permet de réduire la latence entre les clients connectés et le serveur, améliorant ainsi la fluidité du jeu.

Une autre optimisation majeure serait de développer notre propre API en Node.js. Cette architecture, illustrée par la figure 5, ferait de notre serveur API l'émetteur direct des messages WebSocket, évitant ainsi de passer par un prestataire tiers comme Pusher (cf. figure 4).

Cette solution présente plusieurs avantages : tout d'abord, elle permet un envoi plus efficace des messages, ce qui assure une communication plus rapide entre les joueurs. Cela réduit les risques de bugs liés à la latence ou aux téléportations involontaires dans le jeu. Par ailleurs, elle offre un meilleur contrôle sur le traitement des données, car nous ne partageons plus l'intégralité de nos messages avec un service externe sans savoir exactement ce qu'il en fait. Ainsi, nous diminuons notre dépendance à un tiers.

Cependant, l'inconvénient majeur de cette approche est la nécessité de gérer un composant supplémentaire et de développer entièrement notre propre API. En effet, passer à un serveur Node.js implique de réécrire notre code PHP actuel en JavaScript, ce qui représente un investissement en temps conséquent. De plus, Pusher offre des avantages non négligeables en gérant automatiquement les mises à jour et les particularités complexes des réseaux, ce qui nous permettait jusqu'à présent de nous concentrer davantage sur le développement des autres fonctionnalités du jeu.

9.2 Vérification de l'état de la partie depuis le serveur

Un point important sur lequel nous pourrions améliorer la qualité de notre jeu concerne la prise en main des objets et la réalisation des actions des joueurs lors des parties multijoueurs. En effet, dans la version actuelle de notre jeu, chaque client possède une représentation locale de la partie, qu'il met à jour à réception des messages Pusher. Par exemple, lorsqu'un joueur s'approche d'un coffre, le client vérifie localement si un objet s'y trouve. Si c'est le cas, pour offrir une meilleure expérience aux joueurs, le préfabriqué du joueur est immédiatement mis à jour localement afin que l'objet apparaisse instantanément dans sa main.

Comme illustré dans la figure 6, ce graphique montre les actions réalisées en fonction du temps. Prenons l'exemple de deux clients connectés à une même partie, chacun sans objet en main, et avec une épée dans un coffre.

À T+0, l'épée est présente dans le coffre pour tous les joueurs. À T+1, le client 1 décide de prendre l'épée en s'approchant du coffre. Une requête est envoyée au serveur pour informer que l'épée n'est plus disponible dans le coffre. À T+2, le serveur reçoit et traite cette requête. En parallèle, le client 2, ignorant ce changement, s'approche lui aussi du coffre et prend localement l'épée en main. Il envoie également une requête au serveur pour signaler ce changement. À T+3, tous les clients sont informés du changement d'état de la partie : l'épée n'est plus dans le coffre. Le client 1 ne modifie rien car il avait déjà enlevé l'épée localement, idem pour le client 2. À T+4, la requête initiée par le client 2 arrive sur le serveur et est retransmise aux clients, qui ne modifient rien. Aucun crash ne survient, car toutes les requêtes sont valides et traitées correctement. Cependant, à ce moment, on constate que sur le serveur l'épée n'est plus dans le coffre, mais que les deux clients possèdent une épée en main. Nous sommes donc face à ce que l'on peut appeler un "bug de duplication".

Bien que ce problème n'entraîne aucun dysfonctionnement technique ni plantage du jeu, il dégrade considérablement l'expérience de jeu des utilisateurs.

Cette problématique, qui semble simple à corriger de prime abord, s'avère en réalité assez complexe à résoudre dans un environnement multijoueur asynchrone.

Deux méthodes peuvent être envisagées pour éviter la reproduction de ce bug.

Première méthode : Le client envoie une requête au serveur pour signaler qu'il souhaite prendre l'épée. Le serveur vérifie alors en interne si l'objet est toujours disponible. Si c'est le cas, il répond au client qu'il est autorisé à la prendre, puis informe tous les autres joueurs du changement d'état.

Cependant, cette approche engendre un problème : le client doit attendre la réponse du serveur avant de changer son préfabriqué localement, ce qui provoque une sensation de latence et nuit à l'expérience utilisateur.

Seconde méthode : Cette méthode repose sur le même principe, avec quelques différences importantes. Lorsque le client s'approche de l'objet, il le prend immédiatement en main et charge le nouveau préfabriqué localement. Ensuite, il envoie une requête au serveur pour valider cette action. Le serveur vérifie la disponibilité de l'épée à cet endroit et renvoie une réponse HTTP :

Si l'épée est disponible, il informe également tous les autres joueurs du changement. Si l'épée n'est plus disponible, le client reçoit une réponse négative et doit alors retirer l'objet en changeant à nouveau son préfabriqué, simulant la perte de l'épée. Durant ce délai, un enchaînement d'actions un peu étrange peut se produire : personnage sans épée, prise instantanée, puis disparition soudaine de l'objet. Pour atténuer cet effet, il est nécessaire d'ajouter un attribut temporaire sur l'objet pris en main, empêchant le joueur de l'utiliser tant que la réponse définitive du serveur n'est pas reçue.

Avec cette seconde méthode, le joueur bénéficie d'une sensation d'immédiateté tout en conservant une sécurité côté serveur. Le délai, inférieur à une demi-seconde, rend quasiment imperceptible cette contrainte. Cette solution est donc la plus adaptée, car si les joueurs ne cherchent pas explicitement à reproduire le bug, cette disparition soudaine n'arrivera que dans de très rares cas.

À l'inverse, la première méthode entraînerait à chaque prise d'objet une latence visible, répétée de nombreuses fois au cours d'une partie, ce qui dégraderait fortement le ressenti des joueurs.

Ainsi, pour améliorer la performance et la fluidité du jeu, le développement de la seconde méthode devrait être une priorité pour nos futures mises à jour.

10 Conclusion

Pour conclure, après des dizaines d'heures de travail, nous sommes très satisfaits du résultat final de notre jeu. Malgré de nombreux questionnements, échanges et contraintes techniques, nous sommes parvenus à développer un produit qui répond pleinement à nos attentes.

Au-delà des difficultés techniques rencontrées, la coordination entre les membres du groupe a également représenté un véritable défi. Comprendre les besoins, les propositions et les difficultés de chacun n'a pas toujours été simple. Avec quatre membres, les idées se succèdent et s'entremêlent, ce qui a parfois compliqué la définition claire de la direction à prendre.

Les visions du jeu vidéo étant très différentes entre les membres, cela a été l'occasion d'explorer de nouveaux concepts et styles de jeu, enrichissant ainsi notre projet.

Enfin, ce travail, davantage axé sur une vision globale de la création, nous a permis de découvrir de nombreuses facettes impliquées dans la réalisation d'un jeu complet et amusant.

Once Upon a Time est le fruit d'un véritable travail d'équipe, qui nous a permis de produire un jeu de qualité dont nous sommes fiers.

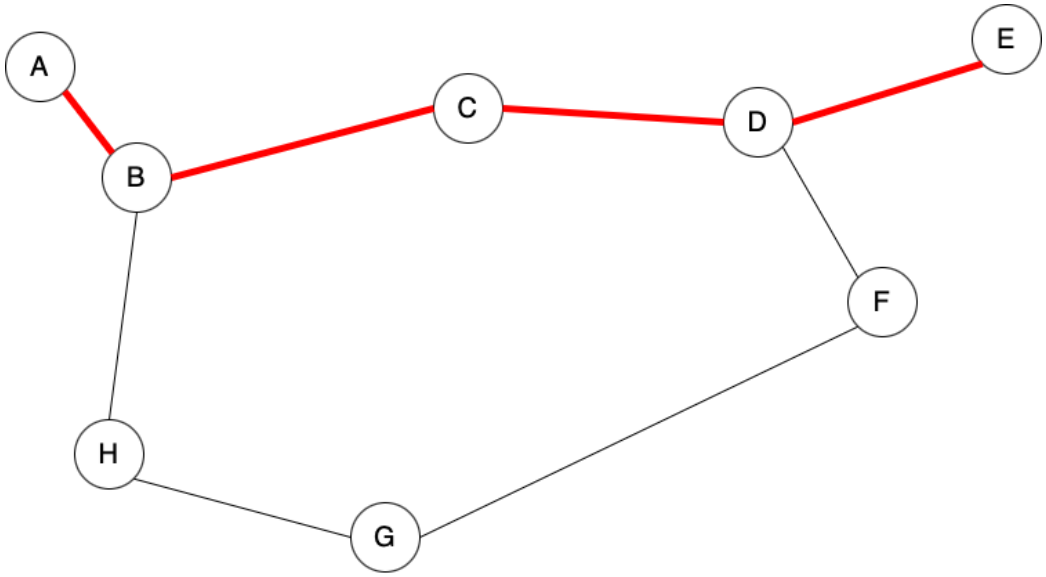

Figure 1: Exemple de graphe pour l'algorythme de Dijkstra

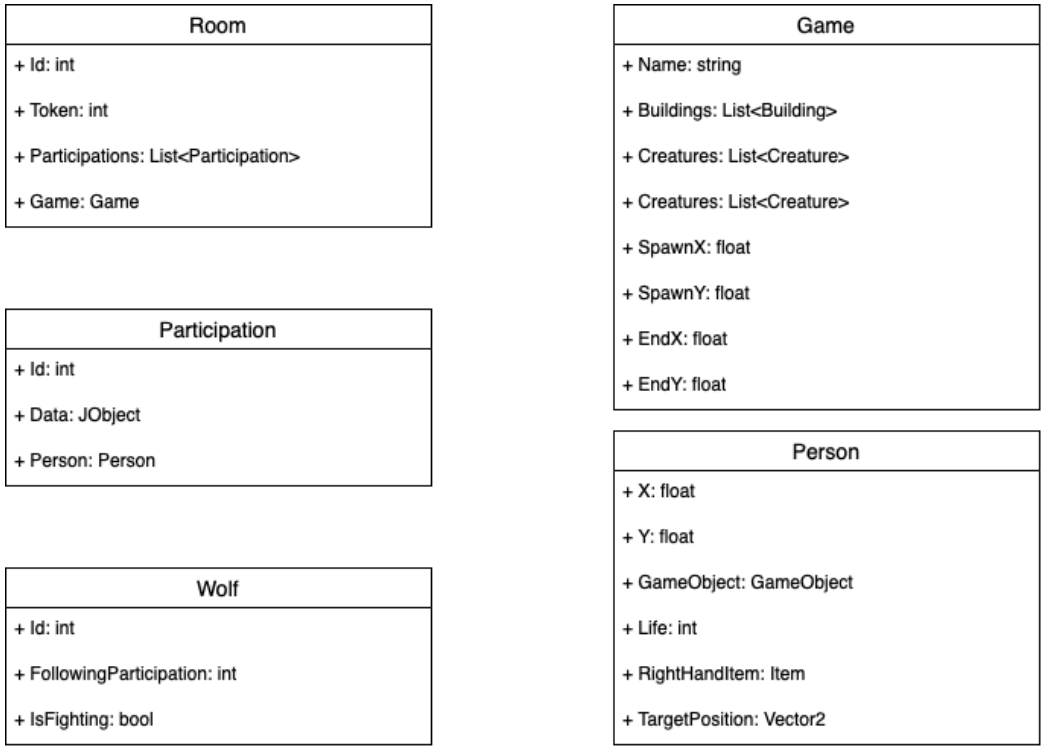

Figure 2: Shéma des différntes classes c# du jeu

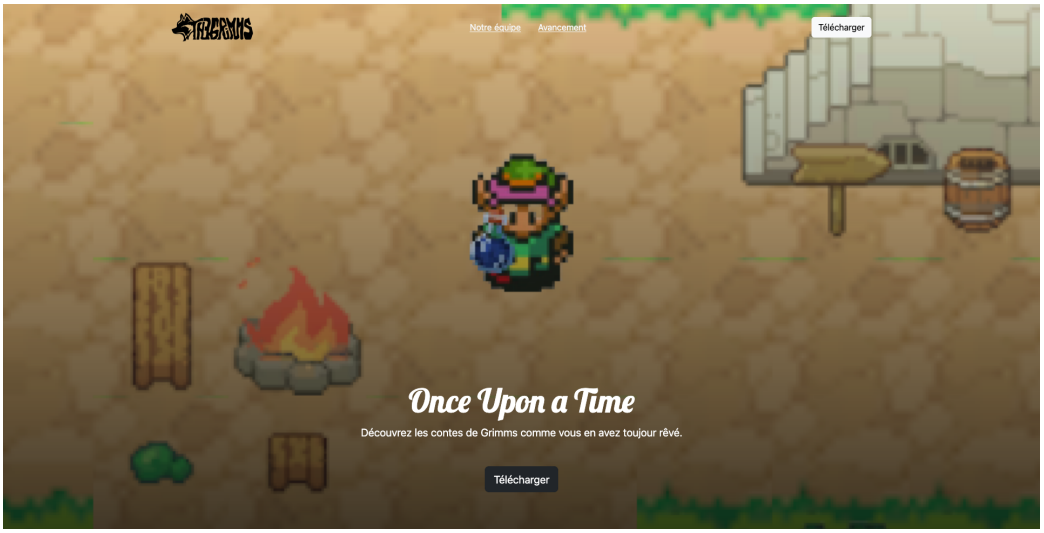

Figure 3: Page par défaut du site web

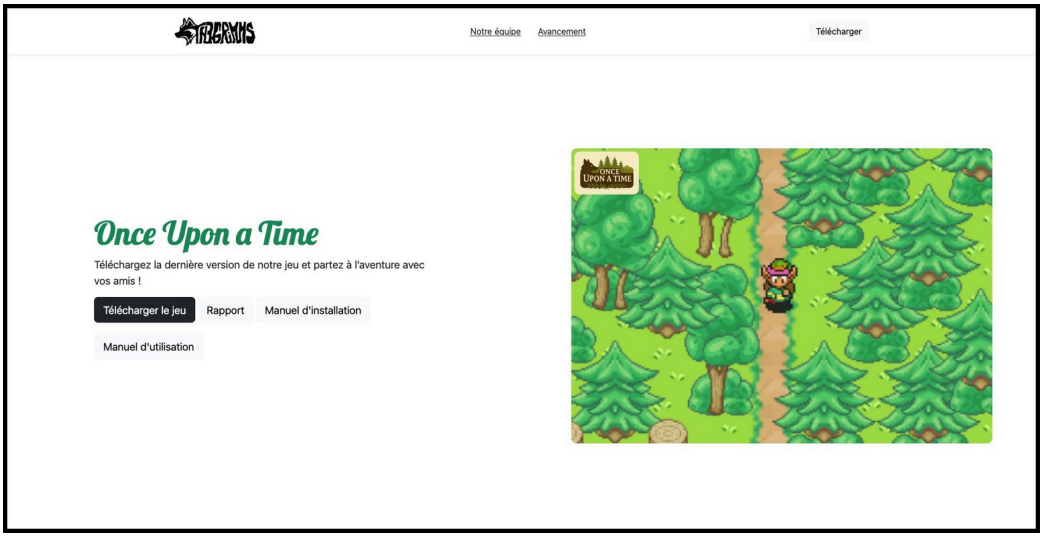

Figure 4: Page de téléchargement de notre site web

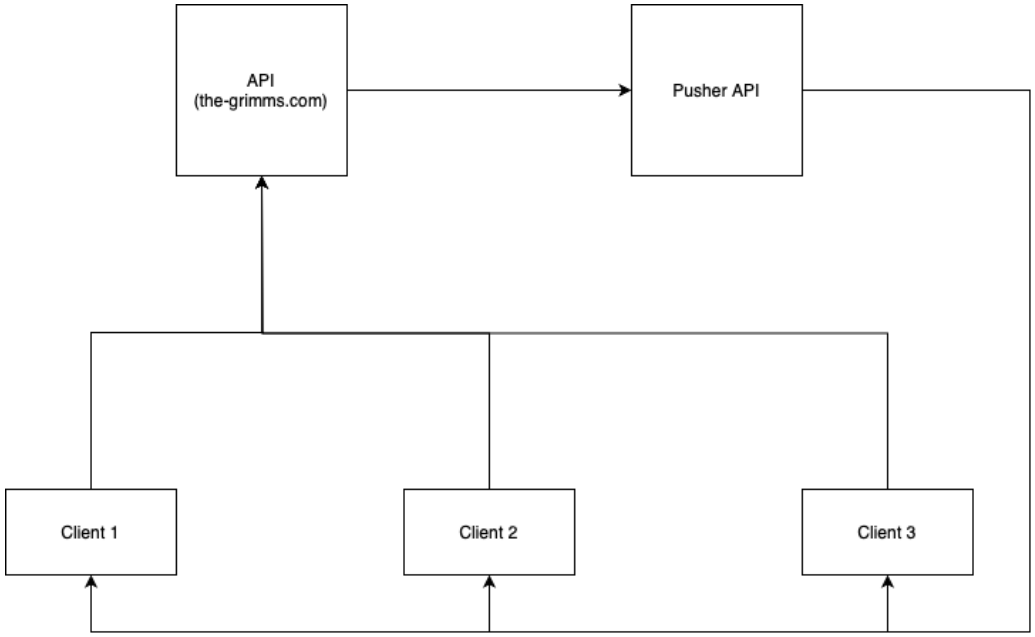

Figure 5: Architecture de notre système de multijoueur

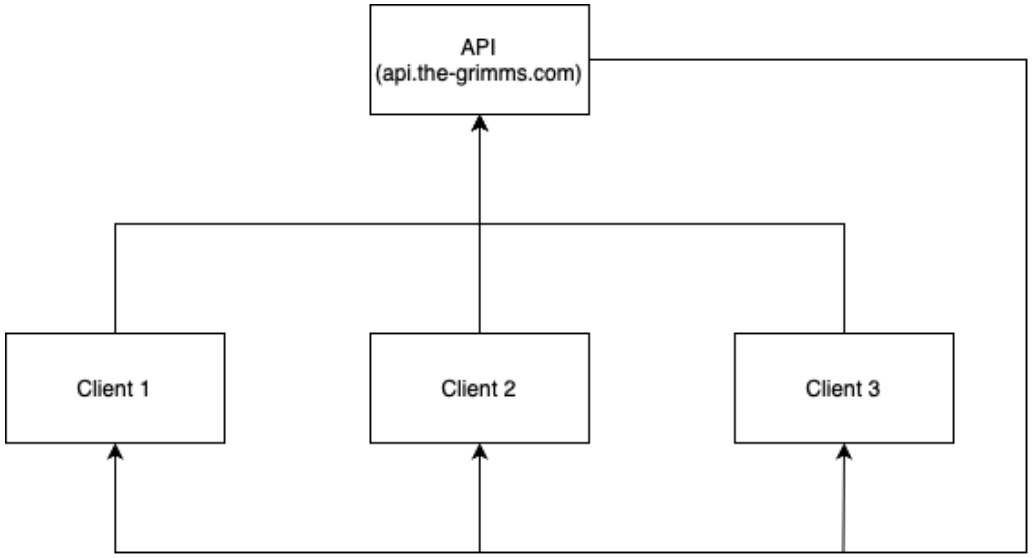

Figure 6: Architecture du système de multijoueur avec un serveur Node JS

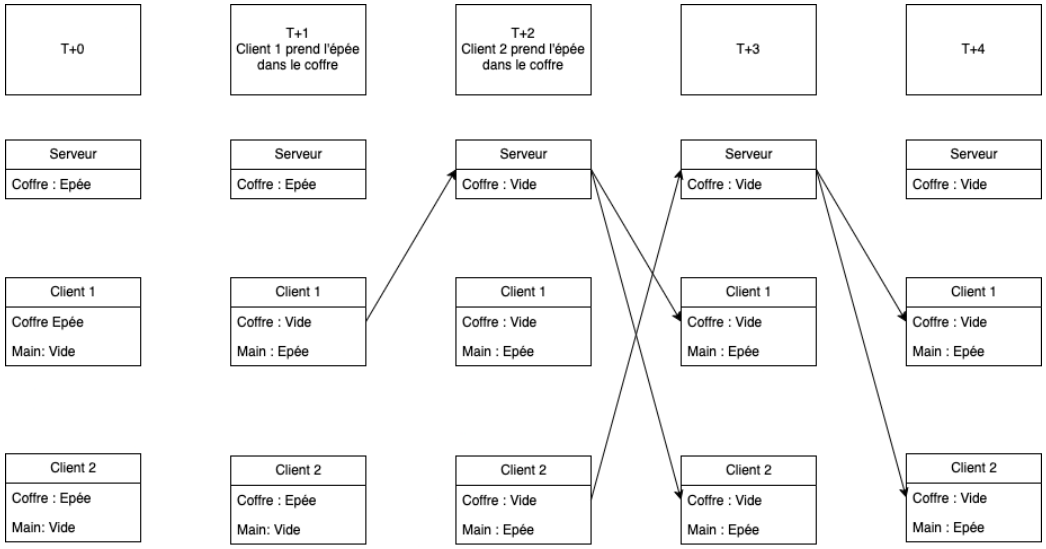

Figure 7: Shéma représentant l'état des clients et du serveur sur une période donnée

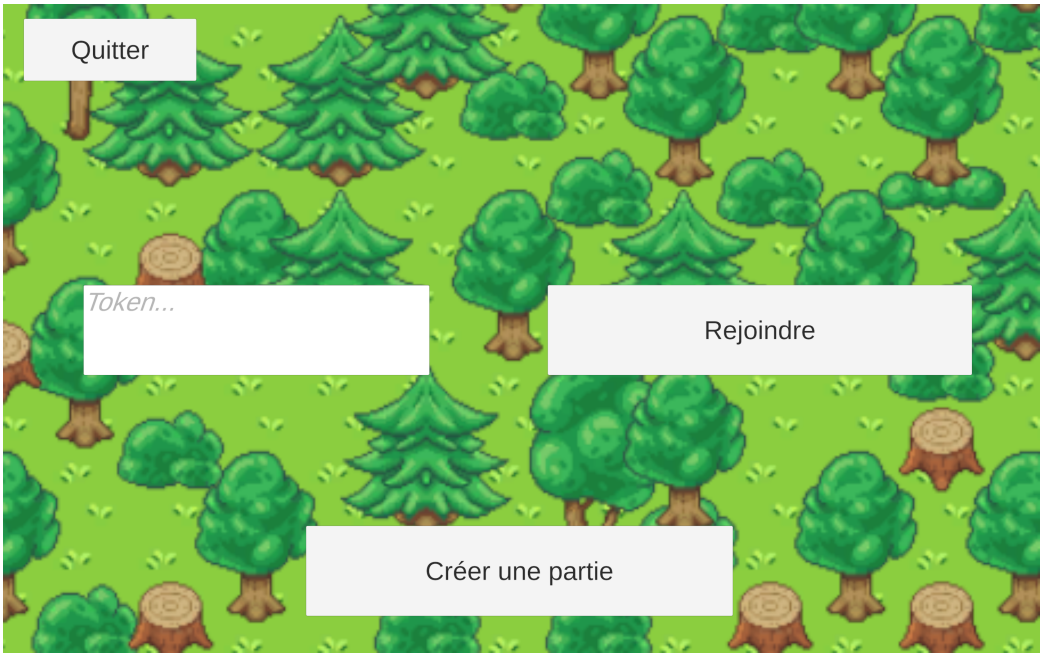

Figure 8: Fenêtre des menus de notre jeu.