Custos Carceris

Seconde soutenance

Deadly Science

Léandre Perrot
Yann Boudry
Steve Suissa
Célian Raimbault

Un projet EPITA

19 avril 2020

Table des matières

1 Introduction **2**

2 Réalisations **3**

2.0.1 Célian 3

2.0.2 Steve 6

1 Introduction

Le jeu a grandement avancé depuis le retour du cahier des charges : le réseau est quasiment entièrement fonctionnel, pareil pour la caméra, le site avance petit à petit, la génération est presque fini et certaines musiques ont été composées. Globalement, le jeu est quasiment jouable.

TABLE 1 – Avancement

Tâches \Soutenances	Attendu	Réalité
Camera	50%	75%
G Joueur	30%	50%
G Jeu	30%	30%
Reseau	50%	75%
Map Const	30%	30%
Menu	15%	70%
Chrono/GUI	30%	30%
Site TXT	0%	30%
Site E	0%	0%
Map Gen	0%	50%
Musique	0%	25%
Sons	0%	10%

2 Réalisations

2.0.1 Célian

Phases et status

Implémentation locale Rappel : Le jeu est composé de deux phases, la première où les joueurs doivent trouver les sérums et la seconde où les joueurs qui n'ont pas réussi à trouver des sérums doivent se venger et empêcher les joueurs guéris de gagner en les touchant.

Un joueur peut avoir 4 status différents, pour l'instant 3 d'entre eux sont implémentés correctement :

- Infécté : Au début du jeu, il faut trouver un sérum pour passer en guéris et non en mode revanche - Guéris : Ce joueur à presque gagné, il doit éviter les joueurs encore infectés - en mode revanche - pour gagner - Revanche : Ce joueur à perdu mais peut empêcher les autres de gagner en les touchant - Fantôme : Ce status n'est pas vraiment implémenté mais servira au début et à la fin du jeu pour attendre et regardé les autres joueurs.

Ces status font parti de l'énumération *PlayerState.PlayerStatus*, il est plus adapté de faire une énumération plutôt que des constantes car les constantes peuvent être mal renseignées.

Lors de la première soutenance, ces status étaient déjà présents mais seulement en local, il était alors primordiale de les synchronisés en réseau.

Implémentation à distance PlayerNetwork est le module présent sur chaque joueur (controlé ou non par le client) qui se charge d'envoyer et recevoir les évenements liés au réseau, il sert également à initialisés les modules présents sur un joueur.

Chaque évenement réseau de PlayerNetwork est décomposée en deux fonctions : *Send<Evenement>* et *<Evenement>*. La première est utilisée par un seul joueur pour appeler la seconde dans toutes les classes PlayerNetwork des joueurs. La seconde est une fonction RPC comme Steve a décrit. La fonction Send ajoute des arguments pour informer qui vient de lever l'évenement, ceci sert à cibler un joueur. Par exemple, nous pouvons envoyer un changement de status au joueur ayant pour identifiant 2 (l'identifiant est un entier unique pour chaque joueur). Pour revenir au status, une fois l'évenement reçu, nous avons juste à trouver si c'est le bon joueur qui reçoit l'évenement puis changer le status.

Les phases sont implémentées de manière semblable. En effet, nous utilisons des évenements comme *BeginGame* lancé au début de la première phase, *EndFirstPhase* lors de la fin de la première phase et *EndGame* quand nous devons gérer la victoire / la défaite du joueur. La seule différence avec le status est que seulement le maître du jeu envoie ces évenements. Cela évite que deux joueurs soient dans deux phases différentes, si un joueur possède un ralentissement ou un bug, la deuxième phase peut durer plus ou moins longtemps. Pour détecter le début du jeu, nous attendons que tous les joueurs soient prêts, ils envoient un

évenement réseau pour avertir le maître du jeu. La fin de la première phase se fait quand chaque sérum est récupéré, là aussi nous avertissons le maître quand un sérum est pris. Après cela, il faut attendre le temps de la seconde phase grâce à une coroutine, c'est une fonction qui s'execute en plusieurs fois, il faut ajouter la ligne si dessous pour attendre quelques secondes : Ensuite il suffit d'avertir les autres joueurs via les évenements réseau.

La fonction *EndGame* est appelée lors de la victoire / la défaite du joueur. Pour savoir si le joueur gagne, il suffit de comparer son status au status guéris, après nous affichons le menu gagné ou perdu.

Endurance et coups

Endurance : Présentation Maintenant nous avons un vrai jeu, c'est-à-dire qu'il y a des joueurs qui peuvent gagner ou perdre et tout cela est en plus synchronisé en réseau. Parlons d'un ajout au jeu le rendant bien plus intéressant ainsi que dynamique : La jauge d'endurance.

L'endurance est semblable à une barre de vie dans la plupart des jeux, contrairement à une barre de vie, une fois vide, elle ne tue pas le joueur mais le rend faible. Ainsi, le joueur se déplace très lentement (10% de sa vitesse normale) et ne peut plus sauter. Cette jauge se remplie petit à petit et une fois le joueur *assomé*, il doit attendre que cette barre d'endurance se remplisse totalement avant de retrouver ses abilités.

A présent, parlons de comment un joueur peut assomer un autre joueur, c'est très simple, à la manière d'un jeu de tir à la première personne, un joueur doit viser une cible puis cliquer avec sa souris. Bien sûr, il y a une portée maximale car le joueur ne tir pas vraiment, c'est juste un coup.

De plus, en seconde phase, si un joueur de status vengeance donne un coup à un autre joueur de status guéris, celui-ci prends le status vengeance. Cette façon permet aux joueurs de plus facilement interragir entre eux.

Endurance : Implémentation Cette fonctionnalité s'implémente en deux temps. Premièrement, il faut détecter les coups entre joueurs ainsi que de gérer le mode assomé. Après cela, il faut tout porter en réseau et ceci comprends également l'affichage de la barre d'endurance au dessus des joueurs.

Pour détecter les coups, j'ai créé la fonction Player.Attack qui est appelée quand l'utilisateur clique. Cette fonction utilise des RayCasts fournis par Unity, ils décrivent une détection de collision par rayon. Il faut spécifier le point de départ du rayon, sa direction, sa distance ainsi qu'un masque qui permet de filtrer les collisions. Ce masque retient seulement la catégorie de joueur. Une fois touché, nous allons appelé un évenement réseau qui va mettre à jour l'endurance voir le status du joueur touché. Nous avons rencontré un problème lors de l'implémentation de l'endurance, au début, à chaque mise à jour de la valeur nous l'envoyons via le réseau pour la mettre à jour de partout. Le problème est qu'elle peut changer 60 fois par secondes lors de la génération, ce qui à pour

effet de faire voler / nager dans le sol les joueurs... J'ai alors fait une fonction qui s'execute tous les tiers de secondes et met à jour l'endurance en réseau.

Un dernier élément que l'on peut ajouter est un affichage de status et d'endurance au dessus de la tête des joueurs. Ces éléments appartiennent aux joueurs non controllés par le client et sont toujours tourné vers la caméra. Il faut alors enlever celui du client au début de la partie et les mettre à jour lors d'un changement de status ou d'endurance.

Gestion de l'audio

Musique et effets sonores Lors de la dernière soutenance, j'avais évoqué la possibilité d'intégrer des sons au jeu si le jeu avait suffisament avancé. Comme c'est le cas nous avons intégrer quelques sons et musiques. Léandre s'est occupé de composer les musiques et pour ma part je les ai mixées et intégrées au jeu. Pour le moment les musiques sont celles du menu et du jeu et nous avons également composés les effets sonores de victoire ou défaites.

Intégration de l'audio Voici comment j'ai penser à intégrer l'audio au jeu. Tout d'abord, les musiques et effets sont des listes regroupant à la fois un identifiant et un fichier audio. J'ai créé une classe Audio premièrement qui possède des fonctions statiques ainsi qu'une instance statique, je me suis inspiré du design pattern *Singleton* pour cette classe afin de rendre plus facile d'accès les méthodes. Cette classe doit être alors unique et pour cela j'ai fait appel à la fonction DontDestroyOnLoad de Unity, elle permet comme son nom l'indique de ne pas enlever l'objet ayant ce script lors d'un changement de scène. Maintenant distinguons deux types d'élément audio : les effets sonores ainsi que la musique. Les effets sonores sont des fichiers audio souvent de quelques secondes, ils interviennent lors de la victoire par exemple, ils peuvent être dans l'espace 3D, c'est à dire, plus on s'éloigne, moins le son est fort et en plus il peut être plus fort à gauche qu'à droite. Ces sons doivent se détruire automatiquement une fois terminés. Quant aux musiques, elles ne se terminent *jamais*, elles se jouent en boucle et il n'y en a qu'une seule à la fois. Avec Unity, ce n'est pas compliqué d'implémenter une fonction *Play* et *SetMusic* qui jouent soit un effet audio, soit de la musique, ce qui est moins simple est de faire des transitions entre deux musiques quand on doit en changer.

Voici la fonction Play simplifiée :

SetMusic est semblable, voici une partie du code permettant de faire la transition entre deux musiques :

Il peut alors y avoir deux musiques qui jouent en même temps. Pour faire cela, j'ai utilisé un Animator, ce composant permet de gérer plusieurs animations sur un objet. Il y a deux animations : celle d'entrée et celle de sortie. Premièrement, j'avais directement animé le volume du composant AudioSource mais nous ne pouvons pas par la suite régler ce volume par les paramètres alors j'ai créé un autre script qui combine deux volumes, celui fournit par l'animation et celui des paramètres, comme leurs valeurs sont entre 0 et 1 il suffit de les multipliés

entre eux pour obtenir le bon son. Pour que les transitions se jouent au bon moment nous déclanchons un *trigger* dans l'animator, c'est juste une fonction qui déclanche une nouvelle animation. Ils se déclanchent au lancement de la musique et lors d'une transition. De plus, ces deux animation durent le même temps pour que ça soit plus agréable.

2.0.2 Steve

Réseau Comme dans l'ancienne soutenance, j'ai toujours la charge du projet et je me charge particulièrement du réseau. L'un des plus grands obstacles qu'on a rencontré lors du réseau est l'identification des joueurs. En effet, il serait bien plus plaisant à jouer si chacun des joueurs pouvait apparaître dans un endroit différent du labyrinthe pour premièrement ne pas avoir d'informations concernant l'emplacement des autres joueurs, et deuxièmement pour ne pas savoir notre propre position dans le labyrinthe. Au début, on avait pensé à faire apparaître chacun des joueurs dans un coin du labyrinthe mais cela pourrait donner des informations sur notre emplacement et pourrait nous faciliter la recherche des sérums.

Finalement, on a décidé de les faire apparaître aléatoirement dans le labyrinthe mais de manière que chacun des joueurs apparaissent à un endroit différent. Ils apparaitront dans une cage et lorsque la partie débutera, une trappe en dessous du joueur s'activera et fera tomber les joueurs dans le dédale. Au premier abord, cela semble très simple mais c'est peut-être l'un des plus grands défis de programmation que j'ai dû relever à ce jour. Des dizaines d'heures ont été passés juste pour cette tâche. On a dans un premier temps, pensé à prendre la liste des joueurs affiché dans le menu précédant, puis à chaque joueur lui assigné un identifiant, et pouvoir ensuite les faire apparaître à des coordonnées précises grâce à ces identifiants. Malheureusement, on a heurté un problème que l'on n'arrivait pas à distinguer le joueur client du reste des joueurs. Malgré qu'il y'ait une variable Player.LocalPlayer (joueur local en français), lorsque que nous avons fait des différents tests, nous nous sommes rapidement rendu compte que le LocalPlayer portait les mêmes informations avec tous les joueurs, c'est à dire même UserID (Identifiant d'utilisateur), ActorNumber (Nombre permettant de déterminer le joueur dans une salle spécifique), et même son pseudo.

Dans un second temps, on a pensé a utilisé les RaiseEvent (Emetteur d'événements) pour pouvoir les envoyer au MasterClient (l'hôte de la partie), pour qu'il serve de base et puisse traiter les différents messages. Cette solution posait deux problèmes : le premier est qu'on ne puisse pas déterminer l'émetteur du signal. En effet, les RaiseEvent possèdent peu de paramètres, on peut soit les envoyer à tout le monde, soit uniquement au MasterClient (il aurait aussi fallu régler le problème que le MasterClient ne s'envoie pas des signaux à lui-même), ou enfin soit à tout le monde sauf lui (encore une fois, si on veut déterminer le joueur émetteur il faut une liste de tous les joueurs et voir lequel des joueurs n'apparait pas dans la liste, mais comme précédemment cela semble trop complexe). Mais le second problème et le plus grand, c'est que lorsque qu'on met dans le code d'envoyer le message, tous les joueurs vont envoyer le même mes-

sage !

Et donc à partir de là, malgré de multiples tentatives de gérer les flux entrants des flux sortants, ce fut bien complexe. On a aussi pensé à utiliser les Photon Views (abrégées PV, moyen de reconnaitre facilement par un identifiant un objet contrôlé soit par la scène, c'est à dire l'hôte, ou bien le client). Le problème c'est que pour mettre une PV, il faut d'abord un objet sur lequel le mettre. Or on souhaite justement à faire apparaître ce dit objet, qui serait notre joueur. On a aussi pensé a utilisé des temps de latence pour envoyer les messages, de manière que s'ils sont envoyés un par un, ils seront bien plus simples à manipuler. Le problème c'est que Photon ne perçoit pas le temps de la même manière que nous le faisons. En effet, même si on essaye de recréer cette latence, Photon va par lui-même gérer cette latence et la partager dans tout le réseau.

Dans la théorie c'est bien, ça permet à ce qu'un joueur avec une bonne connexion n'ait pas d'avantage comparé à celui qui habite dans la campagne, mais dans la pratique et dans notre cas, ce n'est pas ce que l'on souhaite. Après de très longues recherches sur les bas-fonds d'Internet et l'avis de personnes qui en connaissait bien mieux dans la matière que nous. On a opté pour un système utilisant le compte des joueurs dans la partie, car il semblerait que ce soit l'un des seuls attributs de Photon qui dépend réellement du temps (qui a cliqué en premier, qui a la meilleure connexion etc. ...) Et ça fonctionne bien et c'est le principal !

Menus Pour ajouter de la personnification aux jeu, Léandre et moi avons travaillé encore ensemble pour rendre le gameplay et la jouabilité meilleure. L'une des idées qui furent mises sur la table est ce que l'hôte de la partie pourrait choisir les dimensions du labyrinthe à partir d'un menu et pourquoi même pour l'avenir un choix de dimension infini en laissant l'hôte taper ses propres coordonnées pour le labyrinthe.

Pour cela il a fallu créer un menu avec une Scroll List (un tableau déroulant), permettant de sélectionner la dimension que l'on souhaite parmi une multitude de choix. Lorsqu'on veut sélectionner une dimension qui nous convient, il suffit simplement de cliquer dessus, et le travail est joué ! Un script va interpréter l'action puis l'envoyer à l'autre scène pour que le labyrinthe est connaissance de sa nouvelle dimension. Il a aussi refallu changer l'interface du menu de la salle pour l'hôte de manière qu'il ne puisse pas cliquer ou non sur le bouton "Ready" (prêt en français, vu que ce c'est lui qui détermine quand on commence la partie).

En dehors de ça, il a fallu faire les tâches classiques qui sont d'optimiser les scripts, nettoyer les salles des objets de test, repenser les menus pour qu'ils soient plus intuitifs et plus plaisant à regarder...