Custos Carceris

Première soutenance

Deadly Science

Léandre Perrot
Yann Boudry
Steve Suissa
Célian Raimbault

Un projet EPITA

7 mars 2020

Table des matières

1 Introduction

2 Réalisations

2.1 Yann Boudry

2.1.1 Menu Principal

Le menu est la première image qu'aperçoit le joueur du jeu. Il est important d'avoir un menu bien réalisé et qui donne envie de jouer à l'utilisateur.

FIGURE 1 – Main Menu

Menu J'ai choisi de faire un menu simple qui comporte les fonctionalité essentielles que l'on attend de cette partie du jeu. Il comporte donc trois boutons :

- Jouer, qui comme il l'indique permet de lancer une partie.
- Paramètres, qui là encore ouvre une fenêtre de paramètres.
- Quitter, comme son nom l'indique, il arrête l'application.

2.1.2 Jouer

Ce premier bouton vous approche d'un pas de plus vers l'épreuve de survie où tous les coups sont permis : Deadly Science. Après vous être lancé dans l'aventure apparaît un deuxième menu qui regroupe les différents laboratoires abandonnés de Deadly Science. Vous pouvez rejoindre un groupe partant pour une destination connue, ou bien-sûr en chercher un nouveau et l'explorer avec votre équipe. Mais gare aux maladies qui peuvent être... Mortelles !

Si l'aventure vous effraie trop, vous pouvez toujours revenir en arrière pour parfaire la maîtrise de votre personnage au sein des paramètres ou tout simplement quitter le jeu.

2.1.3 Paramètres

Cette partie concerne les paramètres du jeu, pour l'instant ne comprenant qu'un gestionnaire de touches pour les différents mouvements (avant, arrière, gauche, droite, sauter).

Principe Unity propose un gestionnaire de touches mais il faut pour cela avoir le projet et l'éxecuter dans Unity. Il est impossible de changer les touches une fois que le jeu est lancé. J'ai donc cherché des manières de remédier à ce problème.

Implémentation

Rebind Rien de plus énervant que de lancer son jeu favori mais de devoir réassigner les touches de son clavier à chaque fois. Afin d'empêcher toute destruction inopinée de materiel informatique j'ai écrit un script pour cela : InputManager. Il rassemble les touches utilisées et enregistrées, une fonction retournant la/les touches enfoncées, une autre retournant la touche associée à l'action demandée pour l'affichage et le changement de touches.
Pour gérer et enregistrer les touches, j'ai utilisé un dictionaire de string/Keycode auquel j'attribue des valeurs par défaut dès que le script est appelé :

- Jump => Espace
- Forward => W
- Backward => S
- Left => A
- Right => D

J'ai opté pour l'utilisation des PlayerPrefs pour la sauvegarde des touches car c'est une classe très facile à manipuler. Elle ne permet de stocker que des string, float ou int mais la conversion est très simple puisqu'il est possible de convertir les Keycodes en un int unique à l'aide d'un simple cast.
Le script va ensuite tester l'existence de PlayerPrefs sur l'ordinateur de l'utilisateur auquel cas les touches prendront ces valeurs.

```
    for (int i = 0; i < buttonKeys.Count; i++)
        {
            if (PlayerPrefs.HasKey(keys[i]))
                buttonKeys[keys[i]] =
                    (KeyCode) PlayerPrefs.GetInt(keys[i]);
        }
```

Finalement, on sauvegarde les touches. (Utile seulement lors d'un premier lancement du jeu)

En jeu J'ai également du refaire une fonction qui teste quand les touches sont enfoncées. Pour éviter de devoir changer les touches dans tout le code à chaque modification, elle prend le nom de l'action en argument et retourne l'état de la touche correspondante grâce au dictionnaire.

Affichage Le panneau de paramètres utilise un script KeybindDialog qui récupère les noms des touches à partir de l'InputManager. Il va ensuite instancier un préfab pour chaque touche, constitué du nom de l'action et d'un bouton avec la touche correspondante. Lorsque ce bouton est appuyé il va écouter tout le clavier et réassigner la première touche enfoncée à l'action demandée. Le changement est sauvegardé sur l'ordinateur avec les PlayerPrefs en simultané pour la prochaine utilisation.

2.1.4 Pause

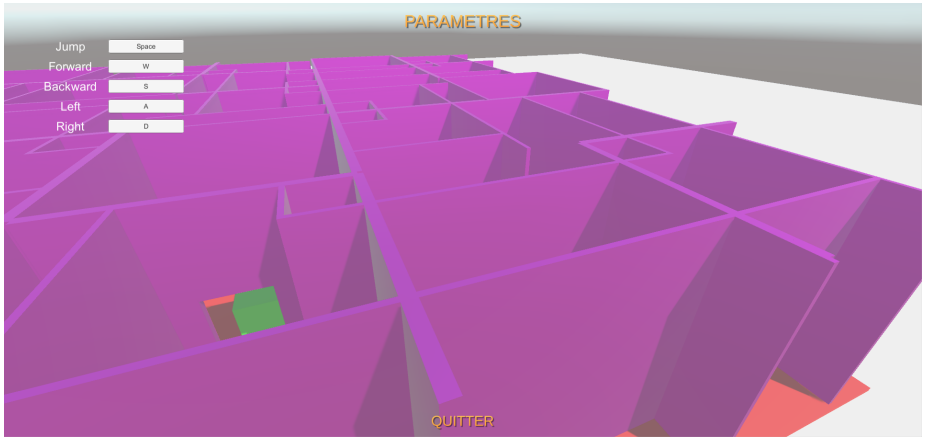

FIGURE 2 – "Pause"

Un jeu n'existe pas sans son menu pause. Enfin... la pause dans un jeu en ligne est discutable. On y retrouve les paramètres pour changer les touches enrichies d'un bouton quitter pour abandonner ses frères d'armes à leur sort et revenir au chaud du menu principal.

2.1.5 Écran de fin

2.2 Célian Raimbault

Dans ce projet, je suis chargé d'implémenter en priorité le comportement des joueurs, la musique et les effets. Pour cette soutenance je me suis concentré exclusivement sur le joueur car la musique et les effets serviront à rendre plus agréable le jeu une fois que nous aurons implémenter la base du jeu, c'est-à-dire prinicipalement le réseau avec le contrôle du joueur.

2.2.1 Gameplay du joueur

Le gameplay du joueur est un des points les plus important du jeu, de plus, il est important de commencer ce projet avec celui-ci. Cette partie se décompose en deux sous parties, la gestion de la physique ainsi que l'intéraction entre le joueur et l'environnement.

Gestion de la physique La physique du joueur est la première chose à implémenter car sans elle il n'y aurait pas de mouvements des joueurs et encore moins d'intéraction. De plus, cette partie doit être réalisée avec minutie car elle peut radicalement changer le gameplay si elle est bien (ou mal) réalisée.

Les physiques de Unity Unity dispose d'un moteur physique, c'est lui que nous utilisons car il est très complêt et adapté pour notre jeu. En effet, ce moteur physique nous permet d'appliquer des forces sur des objets, de tester si plusieurs objets sont en collision et même de définir des couches permettant de filtrer les collisions entre objets. Dans Unity, un objet peut avoir un corps s'il est dynamique (c'est à dire qu'il réagit à des forces extérieures) avec un ou plusieurs boîtes de collision.

Principe En premier temps, j'ai pensé à définir le joueur avec un RigidBody et un boite de collision en forme de capsule, c'est à dire que le joueur est attiré vers le bas par la gravité et a une forme de capsule pour éviter de bouger quand il tourne et pour mieux pouvoir grimper sur des objets en sautant. Pour bouger le joueur, il suffit de tester à chaque mise à jour du jeu si une entrée clavier est appuyée et appliquer une force. Finalement, après avoir implémenté ce mécanisme je me suis rendu compte que Unity disposait également d'un composant nommé PlayerController et qu'il était peut être plus adapté au jeu. En effet, le PlayerController remplace le RigidBody du joueur, il permet d'entièrement contrôler le joueur, c'est à dire que c'est le rôle du développeur de gérer les physiques comme les forces appliquées sur le corps. Le PlayerController prends juste en charge les collisions entre les autres objets pour ne pas passer au travers. Il est préférable d'utiliser ce composant car nous pouvons par exemple enlever la gravité ou les forces de friction quand nous le voulons. Il est utilisé également pour éviter des bugs causés par le réseau, si deux joueurs sont au même endroit, il ne faut pas les éjecter.

Implémentation J'ai ainsi déclaré un attribut *velocity* au joueur qui est sa vitesse et ajoute une force de gravité à celui-ci. À chaque mise à jour du jeu, cette variable s'ajoute à la position du joueur, elle est juste multipliée par le temps qui s'est écoulé entre la dernière mise à jour pour que le mouvement soit fluide.

```
Vector3 movements = x * transform.right + z * transform.forward;
```

Les mouvements se font presque de la même manière qu'avec un RigidBody, quand il y a une entrée active, ou ajoute à *velocity* une force proportionnelle au temps de pression orientée vers la direction voulue. Il y a également une variable *maxSpeed* qui est la limite de vitesse, si le joueur dépasse cette limite il ne peut que freiner son mouvement, c'est à dire que le produit scalaire entre *velocity* et *mouvement* doit être négatif pour mettre à jour la vitesse.

(Voir annexe [3.1.1])

Pour pouvoir sauter, il a fallu tester si le joueur touchait sol. Pour cela, j'ai ajouté un objet vide qui décrit la position de détection du sol comme ça le joueur teste si le sol est présent dans une petite sphère autours de ce point.

```
bool grounded = velocity.y <= .1f &&
  Physics.CheckSphere(groundSensor.position, controller.radius,
      groundMask);
```

Il faut ajouter une force de frottements (provenant de l'air ou du sol) pour que le jeu soit jouable, sinon notre joueur ne s'arrêtera pas. Bien sûr, cette force est appliquée seulement si le joueur n'avance pas.

(Voir annexe [3.1.2])

Il est évident que le joueur n'est pas censé pouvoir grimper sur un autre joueur ou même sauter depuis un objet qui n'entre pas en collision avec lui. C'est pour cela que j'ai mis en place le système de couches physiques. Il y a, en plus de celles fournies par défaut par Unity, les couches : - Player : Seulement pour les joueurs. - Walls : Pour la détection du sol ou des murs. - PowerUps : Ils ne touchent que les joueurs en tant que *trigger*, c'est à dire que le joueur passe néanmoins au travers.

Intéraction joueur / environnement L'environnement comprends les sérums mais aussi les autres joueurs, le comportement des sérums face au joueur est plutôt simple alors commençons par celui-ci.

Sérums Nous avons pensé qu'il serait envisageable d'ajouter des power ups à notre jeu une fois que celui ci sera plus complet, un sérum peut être considéré comme un power up car c'est un objet qui peut se faire collecter par un joueur. Pour éviter de changer beaucoup de script si nous ajoutons des power ups j'ai pensé à utilisé de la programmation orientée objet en créant une classe PowerUp qui est abstraite, elle possède une méthode *OnCollect* qui est appelée lors d'une collision avec un joueur, elle prends en paramètre le joueur et retourne si le power up doit être détruit. Il ne doit pas toujours se détruire car un joueur

ayant déjà collecté un sérum ne peut pas en prendre en autre. Le sérum utilise alors cette fonction dans sa propre classe.

```
protected abstract bool PowerUp.OnCollect(GameObject player);
```

Joueur et réseau Le joueur est un des éléments les plus utilisés en réseau, sa position doit être synchronisée mais pas seulement, il faut prévenir chaque joueur quand un joueur change de status ou même se fait toucher par un autre joueur. Unity utilise des composants attribués aux objets en jeu, nous pouvons en mettre plusieurs pour un seul objet et c'est cela que nous allons faire pour le joueur. Le but est d'enlever les composants inutiles, par exemple, un joueur controllera un seul joueur en jeu donc le module gérant les mouvements du joueur sera retiré pour les joueurs non controllés. Ceci permet également de préparer le joueur dans une unique prefab en ajoutant tous les modules avec les bonnes propriétés. Cette partie est particulièrement compliquée à imaginer quand on ne s'y connait pas bien en réseau, c'est pour cela qu'il est possible que certaines classes soit modifiées lors d'une future soutenance.

Classes du joueurs

— PlayerNetwork : Permet de gérer les événements liés au réseau et également met en place ou retire tous les autres classes du joueur. Ce script est toujours présent sur un joueur.

— Player : Classe permettant de controller le joueur. Utilisée quand le joueur est controllé par le client.

— PlayerState : Classe gérant le status du joueur. La classe est toujours présente, ses attributs se font modifier seulement par le réseau.

— PlayerMaster : Dispose de fonctions gérant le jeu et les phases. Ce module est présent si le joueur est controllé par le client ainsi que le client est le maître de jeu.

— PlayerSlave : Conçue pour recevoir les appels de PlayerMaster. Présent si le joueur est controllé par le client mais, contrairement à PlayerMaster, le client n'est pas le maître de jeu.

PlayerNetwork Il est important que chaque module d'un joueur soit coordonné avec les autres. Par exemple, si le réseau possède un problème et qu'un joueur se connecte en plusieurs secondes, il faut l'attendre, ensuite préparer le jeu (ce qui comprends la génération du labyrinthe) puis enfin lancer la première phase. Il ne faut pas non plus qu'un joueur puisse bouger avant que le réseau soit initialisé. Pour cela, les modules du joueurs n'ont pas de fonction Start ou Update, sauf PlayerNetwork qui possède une fonction Start et Player qui possède une fonction Update. Dans PlayerNetwork.Start, nous appelons chaque fonction StartAfterNetwork de chaque module pour éviter les problème de synchronisation. Chaque joueur instancié s'enregistre via PlayerNetwork.RegisterPlayer pour pouvoir lancé le jeu une fois que chaque joueur est présent.

(Voir annexe [3.1.3])

En addition, lors d'une collision joueur - joueur il faut qu'un seul joueur 'résolve' la collision, c'est à dire qu'il doit décider du nouveau statut de chacun et informer les joueurs par le réseau de ce nouveau statut. Pour implémenter cela, j'ai décider de créer un identifiant unique pour chaque joueur, puis créée une fonction HasPriority qui va décider quel joueur va avoir la priorité suivant cet identifiant.

(Voir annexe [3.1.4])

2.2.2 Gameplay dans le jeu

Le gameplay dans le jeu comprends les phases de jeu.

Phases Voici les phases que peut avoir un joueur : - Infecté : La première phase, le joueur doit collecter un sérum - Guéri : La seconde phase si le joueur a pris un sérum - Corrompu : La seconde phase quand le joueur est celui qui n'a pas collecté de sérum - Fantôme : Cette phase est celle que prendra un joueur ayant perdu, il pourra plus tard voir la partie sans déranger les joueurs en jeu.

Pour implémenter ces phases il suffit de créer une enum dans la classe de joueur avec un setter qui va mettre à jour d'autres composants du joueur comme la couleur du nom et également d'autres objets du jeu, il faut avertir les autres joueurs et le jeu de ce statut.*(Voir annexe [3.1.5])*

Quand un joueur entre en collision avec un autre joueur, si la phase est différente nous devons changer le statut des joueurs pour pouvoir infecter un joueur par exemple. Ce changement de statut a lieu également quand un joueur donne un coup à un autre joueur, seulement, si les phases sont les même, alors nous allons changer l'endurance du joueur.

```
if (sameStatus)
    if (!stunned)
        Stamina -= strength;
```

Endurance Pour que les joueurs puissent interagir entre eux même s'ils sont dans la même phase, nous avons pensé à intégrer une jauge d'endurance. Cette jauge se remplit au fur et à mesure du temps. Quand un joueur donne un coup à un autre joueur celui-ci perds de l'endurance, une fois à zéro, le joueur est comme "sonné", ces mouvements sont ralentis et le saut est désactivé. En étant sonné, un joueur ne peut pas reprendre de coup.

2.2.3 Modèle et animations

Certaines rumeurs indiquant que les examinateurs de projets s'étaient lassés des joueurs en forme de cube lors de la première soutenance nous ont obligé d'implémenter un prototype de modèle de joueur ainsi que des animations allant avec. Pour l'instant, nous avons décidé d'utiliser des ressources crées par d'autres utilisateurs disponibles gratuitement sur l'Asset Store d'Unity. Intégrer cela se fait en deux temps, importer le modèle et l'animer puis ensuite jouer les bonnes animations et transitions à l'aide de scripts.

Animation du modèle Unity dispose d'un système d'animation très pratique, l'Animator. Il est composé d'animations et de transitions. Les transitions sont déclenchées par des conditions sur des variables. Les variables utilisées sont *grounded* et *moving*.

(Voir annexe [3.1.6])

Scripting de l'animation Cette partie est triviale car nous avons seulement à mettre à jour les variables au bon moment. La variable *grounded* se modifie après que nous testons si le joueur touche sol. Quant à la variable *moving*, celle ci est mise à jour une fois que le joueur s'est déplacé et elle est obtenue à l'aide d'une autre variable nommée *tangentSpeed*, c'est la norme de la vélocité projetée sur le plan horizontal, si cette valeur est supérieure à une limite, *moving* est true, sinon false. Il y a une limite car le joueur n'est jamais à la vitesse 0 à cause de la friction (son mouvement peut être infime).

2.3 A faire

Voici les tâches à travailler avant la prochaine soutenance :

— Gameplay du joueur : Le joueur peut se déplacer correctement mais la plupart des événements ne sont pas synchronisés entre joueurs comme le status.

— Gameplay du jeu : Ici aussi, il faudra synchroniser les événements de jeu comme les phases, la partie la plus importante du jeu (sans elle il n'y a aucun gameplay).

— Effets : Cette partie ne me semble pas prioritaire pour le moment mais une fois que les autres tâches auront été avancé, donner un peu plus de vie au jeu en rajoutant quelques effets pourrait s'avérer utile.

2.4 Conclusion

Pour conclure, je pensais pouvoir avancer un peu plus vite pour implémenter les phases du jeu mais il a été difficile de prévoir tous les événements réseau et nous avons préféré éviter les bugs lors de cette soutenance et présenter un jeu propre et sans vrai gameplay. Pour la prochaîne soutenance, le premier but sera d'avoir un gameplay en évitant les mauvaises synchronisations en réseau.

3 Annexes

3.1 Célian

3.1.1 Mouvements

```
// If input and (not max speed or braking)
if (movements.sqrMagnitude > .1 &&
    (tangentSpeed < maxSpeed * maxSpeed * speedFactor *
        speedFactor ||
    velocity.x * movements.x + velocity.z * movements.z < 0))
        velocity += movements * Time.deltaTime * speedFactor;
```

3.1.2 Friction

```
// Friction
if (grounded)
{
    velocity.x -= velocity.x * Time.deltaTime * friction;
    velocity.z -= velocity.z * Time.deltaTime * friction;
}
else
{
    velocity.x -= velocity.x * Time.deltaTime * damping;
    velocity.z -= velocity.z * Time.deltaTime * damping;
}
```

3.1.3 Réseau (A)

```
// Ce code est raccourci

// The player controlled by the client
public static Player localPlayer;

// List of all players (as GameObject)
public static List<GameObject> Players;

// Registers a new player in the players list
public static void RegisterPlayer(GameObject p)
{
    // Update the player list
    players.Add(p);

    if (players.Count == PhotonNetwork.PlayerList.Length)
        // Dispatch events
        OnAllPlayersInGame();
}
```

3.1.4 Réseau (B)

```
// When the player controlled by the client hits another player
void OnControllerColliderHit(ControllerColliderHit hit)
{
    // The player collides another player
    // This player must send the event
    if (net.HasPriority(pNet))
    {
        // This player is infected
        if (state.Status == PlayerState.PlayerStatus.HEALED &&
            pState.Status == PlayerState.PlayerStatus.INFECTED)
            PlayerNetwork.SendPlayerStatusSet(net.id,
                PlayerState.PlayerStatus.INFECTED);
        // The other player is infected
        else if (pState.Status == PlayerState.PlayerStatus.HEALED
            &&
            state.Status == PlayerState.PlayerStatus.INFECTED)
            PlayerNetwork.SendPlayerStatusSet(pNet.id,
                PlayerState.PlayerStatus.INFECTED);
    }
}
```

3.1.5 Phases

```
public enum PlayerStatus
{
    INFECTED,
    HEALED,
    CORRUPTED,
    GHOST
}

public PlayerStatus Status
{
    set
    {
        // Update status
        status = value;

        // Change label
        if (player && player.nameUi)
            player.nameUi.SetStatus(status);

        // Update material / play sound...
        switch (status)
        {
            case PlayerStatus.HEALED:
                Debug.Log("Player has status HEALED");
                break;
            case PlayerStatus.INFECTED:
                Debug.Log("Player has status INFECTED");
                break;
        }
    }
}
```

3.1.6 Animations

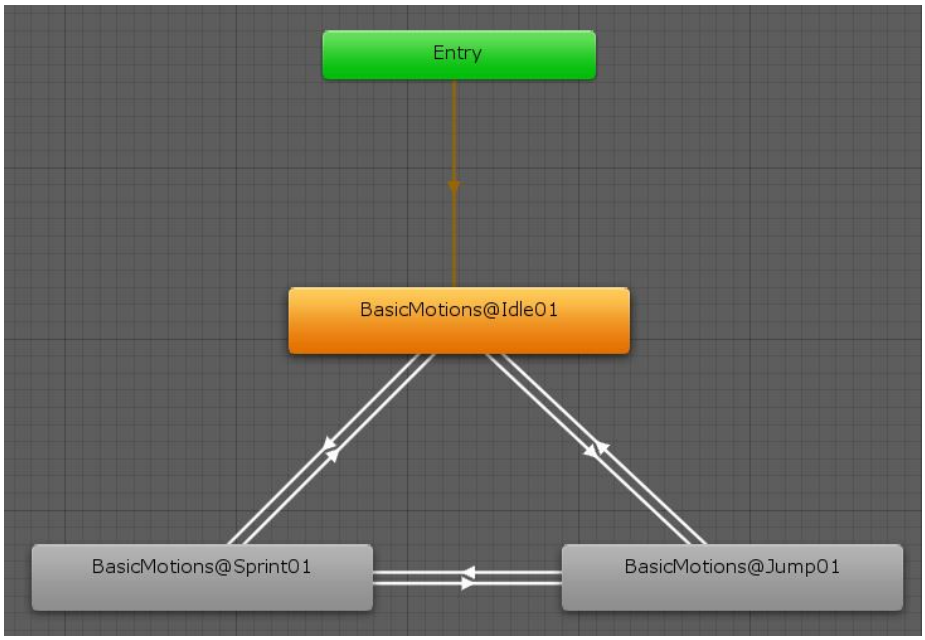

FIGURE 3 – Schéma d'animations