

RAPPORT DE LA PREMIÈRE SOUTENANCE

RELATIF A LA CONCEPTION D'UN JEU DE TIR EN 3D
IMPLÉMENTANT UN MODE MULTIJOUEUR EN
RÉSEAU

“ PATH MAKER”

Paul CREDOZ	E2
Charles-Antoine LEGER	E2
Clément LABBÉ	E2
Pierre-Louis POZZO DI BORGO	E2

SOMMAIRE

<u>I - Introduction</u>	3
<u>II - Réalisations effectuées</u>	4
II.1 - Gameplay	
II.1.1 - <i>Le Player</i>	4
II.1.2 - <i>Déplacement du joueur</i>	5
II.1.3 - <i>Création de la caméra</i>	5
II.1.4 - <i>Lien entre la caméra et le déplacement</i>	6
II.1.5 - <i>Implémentation du tir</i>	7
II.1.6 - <i>Respawn des joueurs</i>	8
II.2 - Mode multijoueur	11
II.2.1 - <i>Gestion des lobbies et prototype</i>	11
II.2.2 - <i>Interaction entre les joueurs dans la partie</i>	13
II.3 - Gestion des comptes des joueurs	16
II.2.1 - <i>Création de la base de données</i>	16
II.2.2 - <i>Création d'un backend</i>	17
II.2.3 - <i>Réalisation d'un prototype de création de compte</i>	19
II.2.4 - <i>Hébergement du backend</i>	20
II.4 - Conception des éléments graphiques	22
II.5 - Modélisation 3D	23
II.5.1 - <i>Modélisation des premiers décors</i>	23
II.5.2 - <i>Modélisation du personnage</i>	24
II.6 - Animations	25
II.7 - Site Internet	27
<u>III - Conclusion</u>	27

INTRODUCTION

L'objet relatif à ce rapport s'inscrit dans le projet du second semestre de la première année d'étude à l'EPITA (Ecole Pour L'informatique et les Techniques Avancées). Ce projet a ainsi pour objectif premier de nous faire découvrir les coulisses de la création d'un logiciel moderne et en particulier celle d'un jeu vidéo en trois dimensions implémentant un mode multijoueur en réseau.

Depuis le rendu du cahier des charges, notre détermination n'a pas baissé et les spécifications de notre projet n'ont pas changé.

Ce rapport est celui de la première des trois soutenance prévu dans ce projet. Dans ce rapport, on détaille les fonctionnalités qui ont été implémentées depuis le rendu du cahier des charges.

GAMEPLAY

Dans cette partie, on présente les fonctionnalités liées au gameplay qui ont été implémentées.

Le Player

Pour ce qui est de la partie gameplay, Clément s'en est chargé avec l'assistance de Pierre-Louis.

Il a d'abord établi le corps du personnage que nous appellerons durant toutes les explications « Player ».

Le corps a donc été ajouté à la scène avec une capsule, qui est généralement utilisée pour faciliter les déplacements du personnage car la capsule est le volume le plus semblable à un personnage humanoïde dans les volumes initialement présents dans Unity.

Il a ensuite ajouté deux petits cylindres au niveau des yeux pour un peu plus de réalisme.

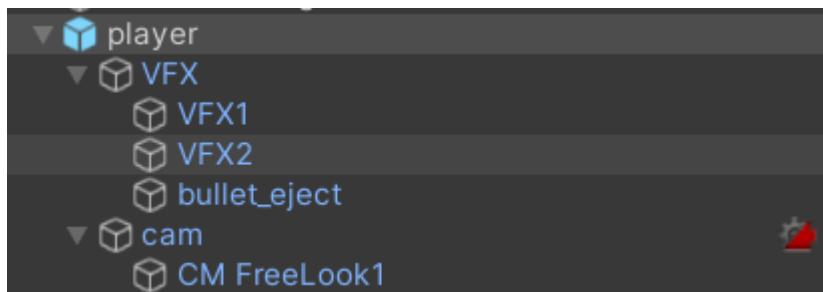
Et surtout pour avoir un repère sur le Player quand celui-ci aura une rotation que nous verrons plus en détails après.

Voici notre Player !



D'un point de vue de Unity, notre Player n'est pas composé exclusivement de corps et de ses yeux nous avons aussi une caméra.

Voici la composition de notre joueur !



Comme vous pouvez le voir ci-dessus ,le Player est composé d'un « empty object » appelé Player, qui contient tous les éléments de notre Player.

VFX est l'ensemble du corps du player, VFX1 et VFX2 sont simplement les yeux du Player. Notre Player possède d'autres éléments que nous verrons plus en détails plus tard.

Déplacement du joueur

Comme dans tout jeux vidéo le Player doit évoluer dans le jeu donc se déplacer et interagir avec son environnement.

Clément a donc ajouté un script permettant de se déplacer dans toutes les directions sur le plan (Oxz) de unity, en d'autres termes sur le plan horizontal. Nous n'avons pas encore réalisé de déplacement sur l'axe Y (sur l'axe Vertical) comme par exemple sauter ou s'accroupir.

Le joueur (personnage contrôlant le Player) déplacera donc le Player avec les touches classiques de Unity en AZERTY soit Z,Q,S et D. ce déplacement respectivement en avant, à gauche, en arrière et à droite.

Création de la caméra

Par ailleurs, notre jeu, Path Maker, est un jeu à la troisième personne, ce qui signifie que la caméra de chaque joueur se situe derrière le corps du Player.

Dans un premier temps, Clément a utilisé une caméra classique de Unity appelée « Main Camera » sur Unity et à paramétrier tous les déplacements de cette caméra. Les déplacements sont effectués avec la souris. Pour faire simple, lorsque le joueur déplace sa souris en avant, en arrière, à gauche ou à droite, le Player va regarder respectivement en haut, en bas, à gauche ou à droite.

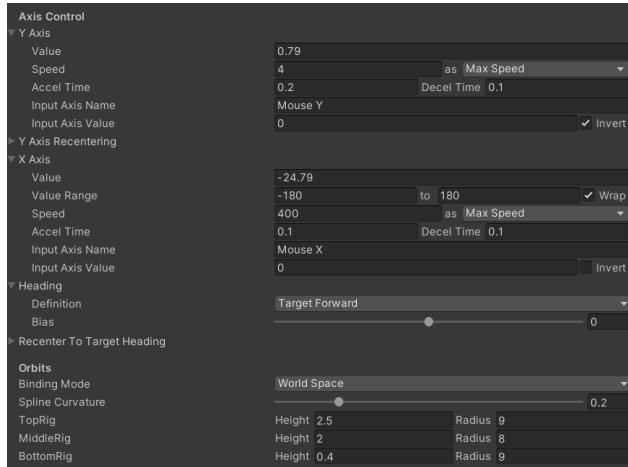
Mais cette « Main Camera » possède un problème : le joueur, en mettant sa souris en avant ou en arrière, fera un tour complet du Player, et pourra être face au Player retourner (après une rotation de 180°). Ce problème est dû à une absence de limite de rotation.

De plus, « l'empty object Player » doit être placé parfaitement au niveau du corps du Player, sinon lors du déplacement de la caméra le corps bougera ce qui pose bien évidemment problème étant donné que le Player doit se déplacer uniquement lors de l'appuis des touches Z,Q,S et D.

Pour pallier ces problèmes, Clément a installé une extension de caméra appelée cinemachine.

Cette caméra contient déjà certaines capacités comme par exemple la possibilité de suivre un objet à une distance paramétrable dans l'inspecteur de Unity.

Voici un exemple des possibilités de paramétrage :



Lien entre Caméra et Déplacement

A présent notre personnage peut se déplacer selon l'axe X et Z, mais ces déplacements ne tiennent pas compte de la position de la caméra. Clément a donc fait un script déplaçant le Player par rapport à la caméra.

```
public CharacterController controller;
public Transform cam;

public float speed = 6f;

public float turnSmoothTime = 0.1f;
float turnSmoothVelocity;

// Update is called once per frame
@ Message Unity | 0 références
void Update()
{
    float horizontal = Input.GetAxisRaw("Horizontal");
    float vertical = Input.GetAxisRaw("Vertical");
    Vector3 direction = new Vector3(horizontal, 0f, vertical).normalized;

    if (direction.magnitude >= 0.1f)
    {
        float targetAngle = Mathf.Atan2(direction.x, direction.z) * Mathf.Rad2Deg + cam.eulerAngles.y;
        float angle = Mathf.SmoothDampAngle(transform.eulerAngles.y, targetAngle, ref turnSmoothVelocity, turnSmoothTime);
        transform.rotation = Quaternion.Euler(0f, angle, 0f);

        Vector3 moveDir = Quaternion.Euler(0f, targetAngle, 0f) * Vector3.forward;
        controller.Move(moveDir.normalized * speed * Time.deltaTime);
    }
}
```

Pour expliquer un peu ce script :

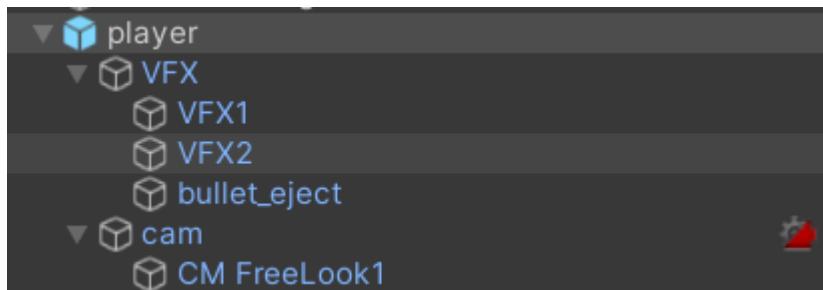
- la variable cam représente notre caméra qui va interagir sur la rotation du Player
- Controller représente un des composants de notre Player qui va le faire déplacer

- les variables horizontal et vertical sont les déplacement du Player par rapport aux touches utilisés par l'utilisateur
- target et angle sont les variables permettant de traduire les mouvement de caméra en rotation
- moveDir est l'ensemble de ces paramètres pour faire avancer le Player en fonction de la caméra

Implémentation du tir

Conformément à un jeu de tir à la troisième personne (vue du dos du Player). Notre Player doit pouvoir tirer !

Pour cela Clément a ajouté un « empty object » appelé bullet_eject comme on peut le voir ci dessous.



Cet élément possède un script appelé “tir” et comme son nom l’indique permet au Player de tirer.

Regardons le script en détails :

```

...
[Serializable]
public GameObject projectil;

private GameObject bullet;

[Serializable]
public float bullet_speed;

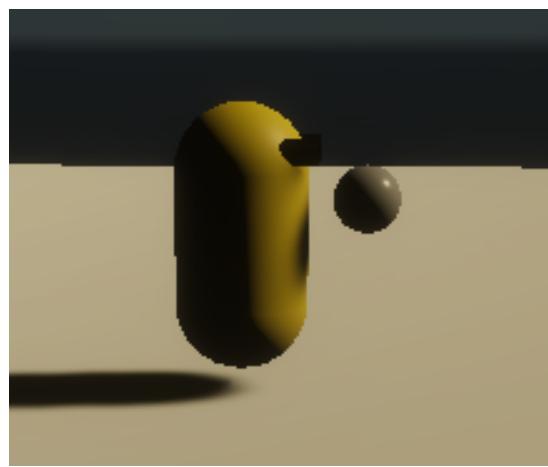
// Update is called once per frame
@ Message Unity | 0 références
void Update()
{
    if (Input.GetMouseButtonDown (0))
    {
        transform.position = transform.position + transform.TransformDirection(Vector3.forward) * 0.1f ;
        bullet = Instantiate(projectil, transform.position, Quaternion.identity) as GameObject;
        bullet.GetComponent<Rigidbody>().velocity = transform.TransformDirection(Vector3.forward) * 0.1f;

    }
    if (Input.GetMouseButtonDown(1))
    {
        bullet.GetComponent<Rigidbody>().velocity *= bullet_speed;
    }
}
}

```

Pour commencer la commande [SerializeField] permet d'afficher dans l'inspecteur de Unity les variables projectil et bullet_speed nous permettant d'y intégrer des “perfabs” (objets) et la vitesse des balles.

- La méthode Start s'appelle juste avant que Update l'appelle pour la première fois.
Dans celle-ci on va simplement déplacer le bullet eject pour éviter toute collision entre le Player et la balle.
- La méthode update s'appelle à chaque image
Quand le bouton gauche de la souris est appuyé, on va faire apparaître une balle avec une vitesse très lente.
Lorsque le bouton droit de la souris est appuyé, la balle va s'accélérer dans la direction initiale de son lancement.



Le Respawn des joueurs

A présent parlons de la réapparition des Players en effet notre jeu utilise ce système pour éviter que la partie ne s'arrête lorsqu'un joueur se fait tirer dessus.

```
④ Message Unity | 0 références
private void OnTriggerEnter(Collider Col)
{
    if (Col.gameObject.tag == "Bullet")
    {
        Debug.Log("Player has been hit !");
        //Destroy(gameObject);
        Respawn(gameObject);
    }
    if (Col.gameObject.tag == "Player")
    {
        Debug.Log("bullet has hit");
        Destroy(gameObject);
    }
}
```

Pour cela Clément a créé un script permettant de faire réapparaître les joueurs, nous nommerons ce processus “respawn”.

Dans le script ci-dessus, nous pouvons observer une première méthode appelée OnTriggerEnter elle permet de détecter les objets en collision avec l'objet portant le script que nous appellerons gameObject.

Explication de la méthode :

Si l'objet en contact avec le gameObject est une “bullet” alors le joueur a été touché et doit donc respawn on appelle donc la méthode Respawn que nous verrons après.

Si l'objet en contact est joueur alors le gameObject est nécessairement une balle donc on la détruit et on affiche dans la console que la balle a touché quelque chose.

Voici la méthode Respawn :

```
public static int spawnPoint = 2;

public GameObject spawnPoint1;
public GameObject spawnPoint2;

private Vector3 respawnLocation;

1 référence
public void Respawn(GameObject player)
{
    int spawnPoints = Random.Range(0, spawnPoint);
    switch (spawnPoints)
    {
        case 0:
            player.transform.position = spawnPoint1.transform.position;
            player.transform.rotation = Quaternion.Euler(0, 0, 0);
            Debug.Log("Repawn1");
            break;

        case 1:
            player.transform.position = spawnPoint2.transform.position;
            player.transform.rotation = Quaternion.Euler(0, 0, 0);
            Debug.Log("Repawn2");
            break;
    }
}
```

On commence par poser le nombre de points de spawn avec la variable spawnPoint (ici il y a 2 points de spawn)

Les variables spawnPoint1 et spawnPoint2 sont des “empty object” dans la scène permettant de donner des coordonnées.

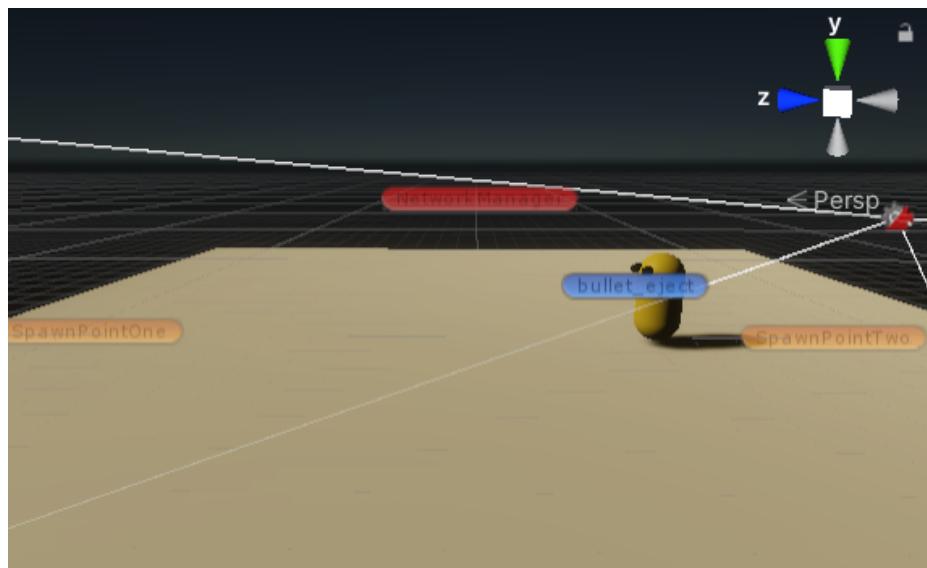
Explication de la méthode Respawn :

On commence par prendre un nombre en 0 et le nombre de points de spawn pour avoir des spawns aléatoires.

Dans le cas 0 : le Player sera téléporté à l'endroit du spawn 1 sans rotation.

Dans le cas 1 : le Player sera téléporté à l'endroit du spawn 2 sans rotation.

On obtient donc la scène suivante :



En rouge au milieu nous pouvons apercevoir le "NetworkManager" nous verrons donc sont utilités après.

MODE MULTIJOUEUR

Dans le planning prévisionnel du cahier des charges, il est dit que la partie concernant le mode multijoueur de notre jeu serait une de nos priorités. Non seulement car elle demande beaucoup de temps (recherches, visionnage et lecture de tutoriels, documentations etc) mais aussi car elle est un des composants majeurs de notre jeu.

On précise dans cette partie les différentes techniques qui ont été et qui seront utilisées tout au long du développement de ce mode de jeu.

Pour permettre à plusieurs joueurs de jouer ensemble, plusieurs systèmes existent dans le monde du jeu vidéo. Les deux systèmes principaux qui existent sont: le système dit avec “serveur dédié”; le système “d’hébergement par le client”.

Pour notre projet, nous avons définitivement choisi le second type d’architecture. En effet, il convient mieux à nos besoins et à notre budget (en effet, un serveur dédié doit être au minimum loué pour le premier type d’architecture évoqué ci-dessus).

Cependant, le système que nous avons choisi pour un problème de taille: comment connecter les joueurs entre eux. En effet, si tous les joueurs jouent sur un même serveur dédié il est plus facile de les connecter entre eux: il suffit de les connecter au serveur (qui ne change pas d’adresse). Le problème qui se pose peut être néanmoins résolu grâce à l’implémentation d’une liste de lobbies affichées dans le menu du jeu et qui permettra aux joueurs de choisir la partie qu’ils veulent intégrer en fonction de plusieurs paramètres comme la localisation géographique du joueur hébergeant la partie, le nombre de joueurs en attente que la partie commence etc. Une fois dans cette pré-partie, les joueurs pourront attendre que la partie commence. Cela permet que les parties commencent toujours avec le bon nombre de joueurs (pour ne pas désavantager une équipe plus qu’une autre).

La gestion des lobbies

Charles-Antoine s'est occupé des recherches liées à cette partie ainsi que de son développement. (avec celui du prototype expliqué ci-dessous notamment).

Il a donc fallu s'intéresser à comment ce genre de systèmes fonctionnent. Pour cela, Unity propose une documentation fournie et détaillée à propos de ce genre de système en général et de leur bibliothèque “Lobby” en particulier. C'est cette dernière qui a motivé notre choix. En effet, cette bibliothèque d'API permettant la simplification de la conception et de l'intégration de ce genre de fonctionnalités convient parfaitement à nos exigences de temps notamment.

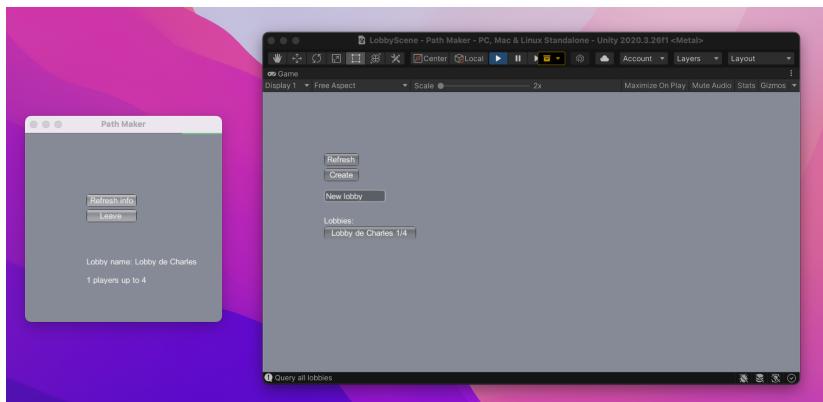
Pour bien comprendre comment il est possible de créer un lobby, d'en maintenir une liste, ou bien d'en intégrer un, nous avons choisi de créer un prototype simple, sans interface particulière mais nous a permis de comprendre les API de la bibliothèque Lobby ainsi que de commencer à se familiariser avec le développement dans Unity (et notamment celui d'un système multijoueur qui complique parfois les tests ou le debugging).

Dans ce prototype, il est possible de voir la liste des lobbies dit “en attente”, de créer un lobby, de voir le nombre de joueurs dans le lobby en cours, d'intégrer un lobby, d'en sortir ou encore de le détruire lors de la sortie du dernier joueur.



Ci-contre, une capture d'écran de la version build du prototype de gestion des lobbies dans “Path Maker”. On y voit plusieurs boutons: le premier, “Refresh” permet d'actualiser la liste des lobbies créés. Ici, il n'y en a aucun.

Nous allons donc en créer un pour l'exemple.



Ci-contre une capture d'écran de la version build du prototype (à gauche) et de la version dans le moteur de jeu Unity (en grand, à droite).

Dans cette capture, on a créé un lobby à partir de la version build du prototype. On est “rentré” dans celui-ci comme l'indique le label: “Lobby name” qui permet aussi de voir le nom du lobby dans lequel nous sommes. De plus, est indiqué le nombre de joueurs dans notre lobby. Ici nous sommes seul sur les quatre joueurs maximum autorisés. On peut aussi voir sur la version qui tourne dans Unity que le lobby appelé “Lobby de Charles” qui vient d'être créé est apparu. On peut aussi voir combien de joueurs sont connectés dans le lobby.

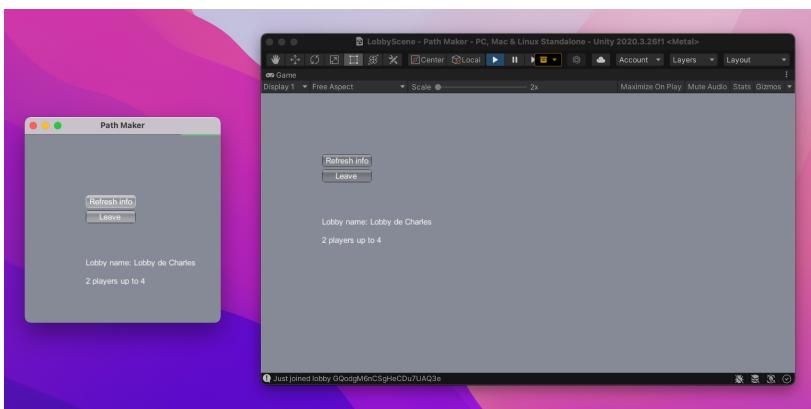
Mars 2022 - S2 - EPITA - YGreg Entertainment

```

private async void CreateLobby(string input)
{
    string lobbyName = "New lobby";
    if (input != "")
    {
        lobbyName = input;
    }
    int maxPlayers = 4;
    CreateLobbyOptions options = new CreateLobbyOptions()
    {
        IsPrivate = false,
        //Player = new Unity.Services.Lobbies.Models.Player(AuthenticationService.Instance.PlayerId)
    };

    Lobby lobby = await Lobbies.Instance.CreateLobbyAsync(lobbyName, maxPlayers, options);
    // Heartbeat the lobby every 15 seconds.
    createdLobbyIds.Enqueue(lobby.Id);
    print("Populate createLobbyIds queue");
    LocalLobby = lobby;
    StartCoroutine(HeartbeatLobbyCoroutine(lobby.Id, 15));
    print("Just create new lobby (code): " + lobby.LobbyCode);
    hasJoined = true;
}

```



Ci-contre, la fonction en C# permettant la création d'un lobby. On utilise ici l'API CreateLobbyAsync de la bibliothèque d'Unity en renseignant le nom du nouveau lobby, le maximum de joueurs qu'il peut contenir ainsi que d'autres options.

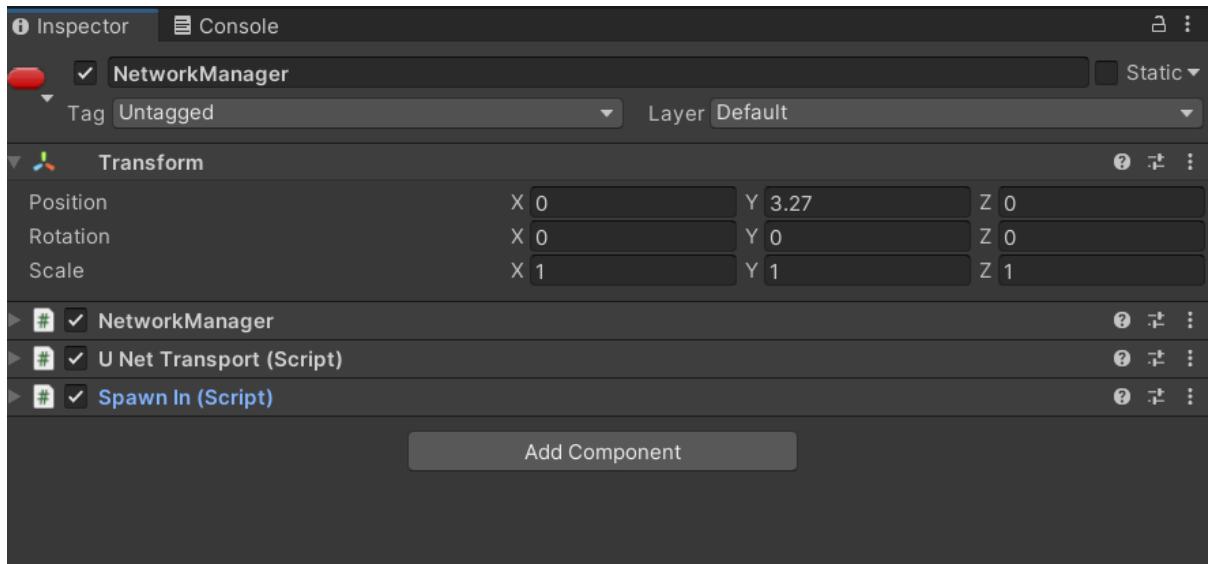
Dans cette dernière capture d'écran on voit que le joueur qui joue dans Unity s'est connecté dans le lobby. En effet, on voit la même interface que pour le joueur qui avait créé le lobby et qui y avait été connecté automatiquement. De plus, on remarque que le compteur des joueurs connectées est passé à deux joueurs sur les quatres maximum.

A la suite de la création de ce prototype, Charles-Antoine a commencé à implémenter ces fonctionnalités dans le projet principal. Il a notamment intégré une interface pour l'utilisateur plus agréable, ce qui a permis au projet de prendre de l'avance sur cette partie qu'est l'interface utilisateur. Ainsi, ont été implémentés un début de menu principal pour le jeu ainsi que pour la gestion des lobbies.

Interaction entre les joueurs dans la partie

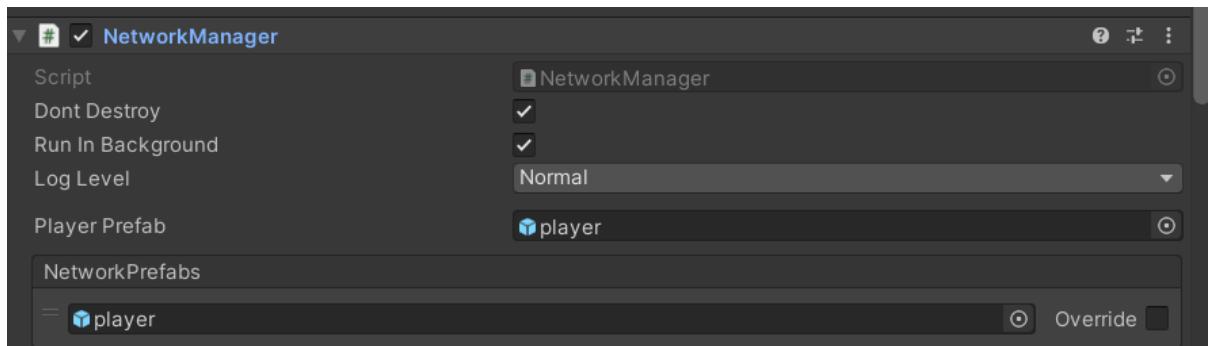
Netcode pour GameObjects est un package Unity qui fournit des capacités de mise en réseau aux workflows GameObject & MonoBehaviour. Le cadre est interopérable avec de nombreux transports de bas niveau.

On a observé précédemment le NetworkManager sur la scène de unity mais expliquons concrètement à quoi il sert.



Le NetworkManager est composé de 3 scripts :

- Le premier des scripts est directement applicable depuis la bibliothèque de Unity Netcode. Il va permettre lors de l'appui sur un des boutons Host ou Client, d'ajouter à la scène un objet donné, ici un player.



- Le second est le transport utilisé lors des différentes interactions .

- D'un script Spawn In qui va afficher lors de la connexion de joueur en haut à gauche le mode de connexion. Dans le cas présent, le joueur décide de se connecter en tant que client ou que host.

A présent nos joueurs peuvent se connecter avec ces deux boutons présents mais un problème est apparu, lors de l'appui sur les touches de déplacement à savoir Z,Q,S,D tous les joueurs présents dans la scène se déplaçaient en même temps car chacun des scripts présents sur les Player étaient actifs.

Clément a donc créé un script permettant de désactiver tous les composants des autres Player.

```

Script Unity | 0 références
public class PlayerSetup : NetworkBehaviour
{
    [SerializeField]
    Behaviour[] componentsToDisable;

    Camera mainCamera;

    Message Unity | 0 références
private void Start()
{
    if (!IsLocalPlayer)
    {
        // boucle pour suppr les composants des autres joueurs
        for (int i = 0; i < componentsToDisable.Length; i++)
        {
            componentsToDisable[i].enabled = false;
        }
    }
    else
    {
        mainCamera = Camera.main;
        if (mainCamera != null)
        {
            mainCamera.gameObject.SetActive(false);
        }
    }
}

Message Unity | 0 références
private void OnDisable()
{
    if (mainCamera != null)
    {
        mainCamera.gameObject.SetActive(true);
    }
}
}

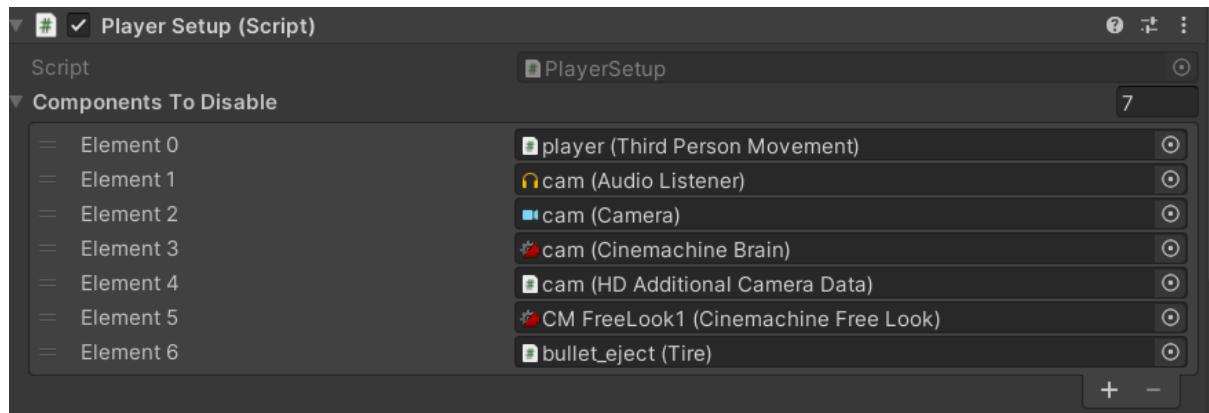
```

On crée en premier lieu une liste des éléments à désactiver (ils seront ajoutés dans l'instructeur de Unity), cette liste est appelée `componentsToDisable`

Dans la méthode Start nous posons une première condition : si le Player est le player local, soit si le player est contrôlé par notre utilisateur. Si notre Player n'est pas local alors on désactive tous ses composants avec une boucle.

Par la même occasion se proceder nous aidera à avoir une seule sortie audio par player car oui beaucoup de composants dans unity sont initialement dotés d'une sortie audio.

Nous avons donc un inspecteur comme ci-dessous :



On désactive les mouvements des autres Players, leur audio, leurs caméras et l'éjection de balle (le tire)

GESTION DES COMPTES DES JOUEURS

Path Maker est un jeu qui se joue en mode multijoueur. Il y a deux possibilités pour ce genre de jeu. La première c'est faire en sorte que les données de jeu de chaque joueur (points d'expériences, nom, meilleurs scores etc) soient stockées en local sur l'ordinateur de chaque joueur. Le principal désavantage de cette méthode c'est que si un joueur veut jouer sur un autre appareil il doit recommencer à zéro sa partie. Pour pallier cela, il est possible de lier à chaque joueur un compte qui lui permettra de récupérer ses données de partie sur n'importe quel appareil voulu.

C'est Charles-Antoine qui s'occupe de toute cette partie.

Choix de la base de données

Premièrement, il a fallu choisir définitivement la base de données avec laquelle nous garderons les données de chaque joueur.

Il existe de nombreuses bases de données pour ce genre d'utilisation. Cependant nous avons vite sélectionné certaines d'entre elles pour les avantages qu'elles apportent. Notre regard c'est d'abord porter sur la base AWS qui propose beaucoup d'offres (gratuites notamment) et qui, par l'abondance des ressources sur internet liées à celle-ci aurait pu être notre choix. De plus, la base Firebase, qui, elle aussi est gratuite pour une utilisation moyenne et elle aussi est assez facile à prendre en main aurait pu être celle choisie. Néanmoins, comme dit dans le cahier des charges, nous avons décidé, et maintenant de façon définitive de rester sur la base de données que nous avions choisi: Mongo DB. En plus d'être gratuite cette base est particulièrement adaptée à nos besoins dans le sens qu'elle est simple d'utilisation et qu'elle s'intègre bien à Unity.

Depuis la remise du cahier des charges en janvier, nous y avons créé un compte ainsi que deux collections (endroit dans lequel seront stockées les données des joueurs et des sessions de ceux-ci.)

A screenshot of the MongoDB Cloud interface. The left sidebar shows 'Charles-Antoine's ORG - 2021-12-02 > MUSICAL-LAMP > DATABASES'. The main area shows the 'musical-lamp' database with '1 DATABASE' and '2 COLLECTIONS'. One collection is 'sessions' and the other is 'users'. The 'users' collection is selected, showing 'STORAGE SIZE: 3KB', 'TOTAL DOCUMENTS: 1', and 'INDEXES TOTAL SIZE: 7KB'. It has tabs for 'Find', 'Indexes', 'Schema Anti-Patterns', 'Aggregation', and 'Search Indexes'. Below these tabs is a search bar with the placeholder 'Find { "Field": "Value" }' and buttons for 'OPTIONS', 'Apply', and 'Reset'. At the bottom, there is a 'QUERY RESULTS' section with a table header 'Document' and a single row with a 'View Document' button.

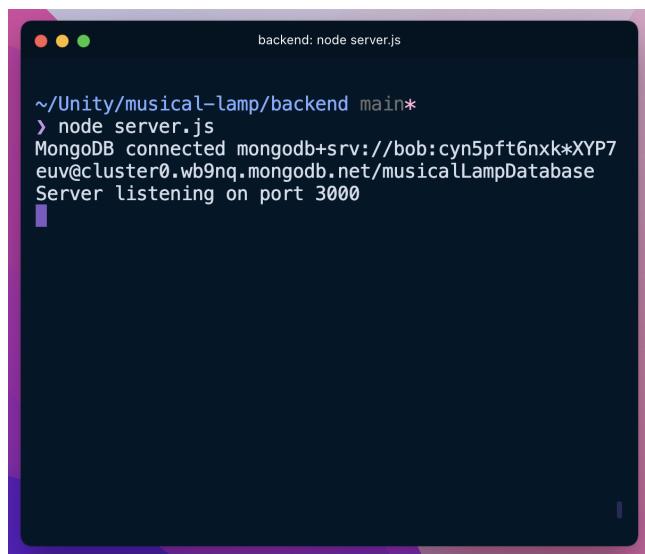
Cette capture d'écran présente le dashboard de la base Mongodb. Plus précisément il s'agit de celui lié au prototype évoqué ci-dessous dans le nom est "musical-lamp". On y voit la collection users ainsi que des sessions.

Création d'un backend

La difficulté de ce système est dû à la combinaison et à la liaison des différentes parties qui la compose. En effet, la base de données est liée à un serveur qui est lui-même lié à Unity.

Il a ainsi fallu créer le serveur (que nous appellerons par la suite “backend”). Ce serveur permet de recevoir les requêtes HTPP d’Unity, de les comprendre et de faire lui-même une requête vers la base de données pour enfin renvoyer au client Unity les informations qu’il souhaitait auparavant. Il est possible de se demander si un tel serveur est nécessaire, il complique en effet le développement du jeu ainsi que son maintien (nécessité d’héberger un tel serveur). Mais après des tests, ainsi que des recherches ont se rend compte que certaines données sensibles comme les identifiants pour se connecter à la base de données ne pourraient pas être stockées dans chaque client Unity. En effet, cela poserait d’importants problèmes de sécurité. De plus, un serveur séparé du jeu lui permet de pouvoir modifier certaines fonctionnalités sans avoir à changer tous les clients de tous les joueurs. Enfin, il est plus facile de s’occuper de certaines tâches (comme l’authentification des joueurs) dans le serveur plutôt que dans le client. D’un point de vue sécurité aussi cela semble logique.

Pour la conception de ce serveur nous avons, comme il est précisé dans le cahier des charges Node.js, express.js et passport.js. Node.js permet de faire tourner du code javascript sur un serveur. Nous parlerons ensuite de l’hébergeur que Charles-Antoine a choisi pour le backend. Express.js est une bibliothèque javascript qui permet la simplification de la création de serveur qui ne font que des requêtes HTTP ou presque. Enfin, passport.js permet l’implémentation de nombreuses “stratégies d’authentification” dont celle avec un nom d’utilisateur est un mot de passe associé.

A screenshot of a terminal window titled "backend: node server.js". The window shows the command "node server.js" being run, followed by logs indicating a MongoDB connection and the server listening on port 3000.

```
~/Unity/musical-lamp/backend main*  
› node server.js  
MongoDB connected mongodb+srv://bob:cyn5pft6nxk*XYP7  
euv@cluster0.wb9nq.mongodb.net/musicalLampDatabase  
Server listening on port 3000
```

Sur cette capture d’écran, on peut voir un terminal faisant tourner node ainsi que le serveur backend.

On peut y lire les log attestant du lien effectué entre le serveur et la base mongodb.

De plus, le message “Server listening on port 3000” indique que le serveur tourne en local, ici sur le port 3000.

C’est dans ce terminal que seront affichés les logs (lors de la création d’un nouveau compte par exemple).

Dans le serveur backend, on s’occupe notamment de renseigner les informations liées aux joueurs. Ci-dessus est présentée une capture d’écran du modèle de création de chaque joueur dans la base de données (il est assez basique pour l’instant mais permet d’apprendre l’utilisation d’une telle base).

```

User.js -- backend
server.js M User.js M X
models > User.js > [?] <unknown>
22 const mongoose = require('mongoose');
21
20 // Create Schema
19 const UserSchema = new mongoose.Schema(
18 {
21     playername: {
20         type: String,
19         required: true,
18         unique: true
22     },
21     password: {
20         type: String,
19         required: true
22     },
21     date: {
20         type: Date,
19         default: Date.now()
22     }
21 },
20     // { strict: false }
21 )
22
23 module.exports = User = mongoose.model("users", UserSchema);

```

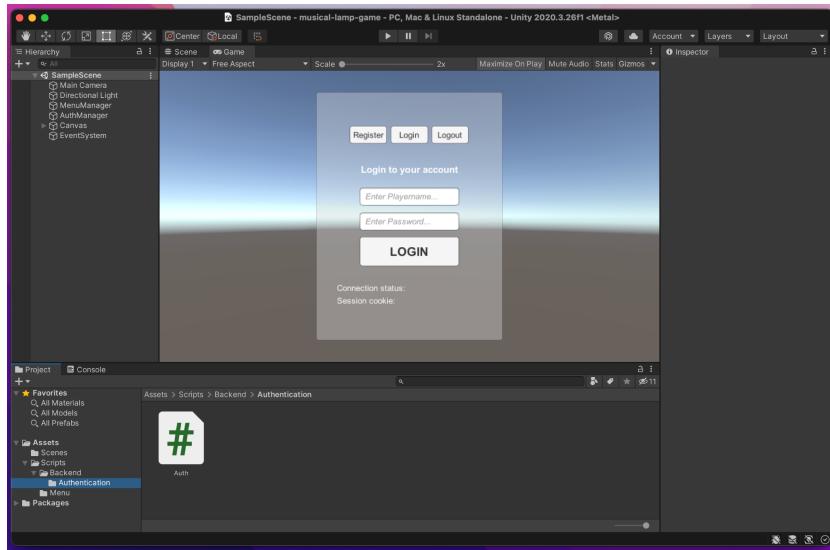
Comme dit précédemment, on s'occupe aussi de la connexion de chaque joueur à la base grâce à la bibliothèque passport.js. Ci-dessous la fonction permettant une telle authentification grâce à l'api passport.authenticate et à la stratégie “local” (qui signifie avec un nom de joueur et un mot de passe, pas de double authentification ou autre).

```
// User Login
auth.post('/login', (req, res, next) => {
  passport.authenticate('local', function(err, user, info) {
    if (err) {
      return res.status(400).json({ errors: err });
    }
    if (!user) {
      return res.status(400).json({ errors: "User not found.", info: info.message });
    }
    req.logIn(user, function(err) {
      if (err) {
        return res.status(400).json({ errors: err });
      }

      return res.status(200).json({ success: `Logged in ${user.id}, your session ID: ${req.session.id}` });
    });
  })(req, res, next);
});
```

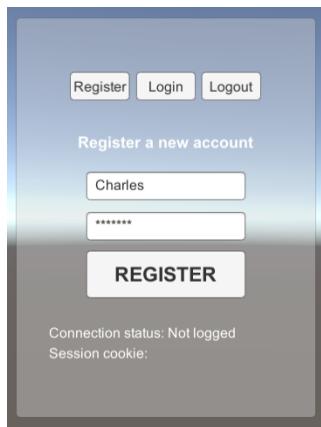
Réalisation d'un prototype de création de compte

Pour comprendre comment nous allons implémenter ce système de gestion de compte, nous avons réalisé un prototype. Dans celui-ci, un joueur peut enregistrer un nouveau compte, se connecter à un compte déjà existant, télécharger depuis la base ses données de jeu et enfin, se déconnecter de son compte. On présente ci-dessous des images de ce prototype ainsi que les explications liées à celles-ci.



La capture d'écran ci-contre présente le cadre de développement du prototype. Il s'agit de l'interface du logiciel Unity. Au milieu est présente l'image qui sera affichée aux joueurs. Sur celle-ci, on peut voir trois boutons: "Register", "Login" et "Logout". Ainsi que deux champs: l'un pour le nom du joueur et l'autre pour son mot de passe. Enfin, un dernier bouton "LOGIN".

On propose de créer un nouveau joueur et de se connecter avec ses identifiants pour l'exemple.



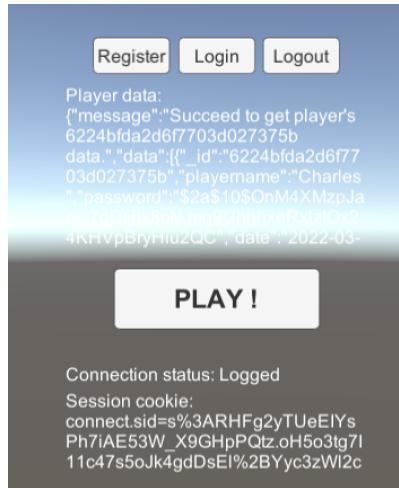
En lançant le prototype, on peut remplir les informations que l'on souhaite données. Dans cet exemple, on choisit de créer un nouveau joueur appelé.

La capture d'écran ci-dessous indique que le nouveau joueur a bien été enregistré dans la base de données grâce au serveur backend. On voit aussi que le mot de passe de Charles n'apparaît pas en clair mais est bien chiffré.

```
QUERY RESULTS 1-1 OF 1

_id: ObjectId("6224bfda2d6f7703d027375b")
playername: "Charles"
password: "$2a$10$0nM4XMzpJaew7gDHjy8cN.mq9UhhxeRxlzI0x24KHVpBryHiu2QC"
date: 2022-03-06T14:05:41.085+00:00
__v: 0
```

Une fois que le joueur appuie sur le bouton “REGISTER”, il arrive sur le menu suivant:



Ce menu indique que l'enregistrement et que la connexion du joueur s'est bien effectuée. En effet, on retrouve sur les informations sur le joueur qui sont stockées dans la base de données, comme son identifiant de base données, ou bien son mot de passe chiffré

De plus, on peut voir que le statut de connexion (en bas de l'image) est désormais “logged” signifiant que le joueur est bien connecté.

Enfin, on voit que le cookie de session (identifiant propre à chaque session de jeu, c'est-à-dire à chaque nouvelle connexion) est complété. Ce numéro permet de ne pas demander à chaque fois au

joueur de se connecter depuis le même ordinateur s'il ne s'est pas déconnecté. En effet, le cookie de session est sauvegardé dans la base de donnée, il contient des informations sur l'ordinateur depuis lequel s'est fait la connexion. Il est aussi stocké en local sur l'ordinateur du joueur.

QUERY RESULTS 1-1 OF 1

```
_id: "RHFg2yTUeEIYsPh7iAE53W_X9GHpPQz"  
expires: 2022-03-20T14:07:06.394+00:00  
session: {"cookie": {"originalMaxAge": null, "expires": null, "secure": false, "httpOnly": true}}
```

L'image ci-dessus indique que le cookie de la session nouvellement créé lors de la connexion de Charles a bien été enregistré sur la base de données.

Hébergement du backend

Comme cela a été expliqué ci-dessus, le système de gestion de compte des joueurs nécessite d'avoir un serveur auquel tous les clients de tous les joueurs peuvent accéder. Il a donc fallu faire des recherches pour savoir sur quelle plateforme nous allons pouvoir héberger un tel serveur.

Après plusieurs recherches, deux hébergeurs semblent être particulièrement adaptés à nos besoins. Le premier est Netlify. En plus d'héberger le site web du projet, Netlify est un bon candidat pour l'hébergement du backend du jeu. En effet, il propose des offres gratuites ainsi qu'une interface claire et moderne. Il permet aussi le déploiement automatique du backend lorsque l'on commit une changement sur le repository GitHub associé, ce qui facilite les mise à jour notamment.

Cependant, Netlify semble être plus adapté à l'hébergement du site web (comme le site du projet) plutôt que de celui d'un serveur ne nécessitant aucun rendu graphique.

Le second hébergeur que Charles-Antoine a sélectionné est Vercel. Vercel est encore mieux adapté à notre projet tant il permet l'hébergement de simple serveur Node.js, ce qui correspond totalement à notre architecture. Nous avons donc définitivement choisi d'entre faire l'hébergeur de ce serveur backend. A l'heure actuelle, Charles-Antoine teste le déploiement du backend sur cette plateforme.

CONCEPTION DES ÉLÉMENTS GRAPHIQUES

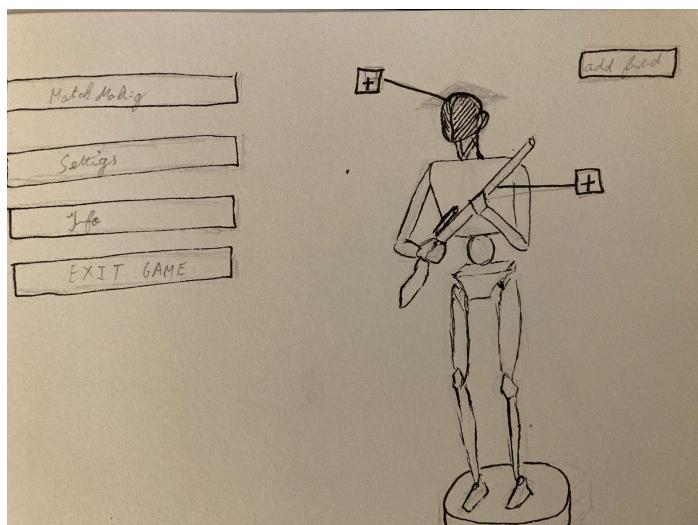
Les différents designs des objets, des personnages et de l'interfaces sont imaginés par la totalité des membres de l'équipe, dessinés par Pierre-Louis pour avoir une bonne idée de ce à quoi ils ressembleront.

Les dessins sont réalisés soit sur papier soit avec une tablette graphique sur le logiciel Clip Studio Paint.

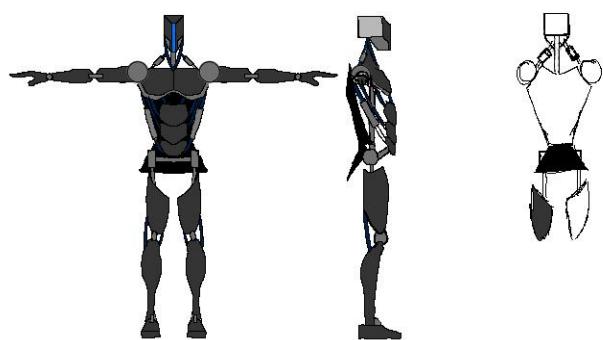
Ensuite, ces différents dessins sont modélisés en 3D grâce à l'outil Blender, nous nous sommes formés grâce à des tutoriels sur Youtube.

L'interface sera quant à elle réalisée sur Figma, plateforme de création de design d'interface.

Voici un des dessins représentant la forme globale de l'interface du menu.



Et ceci est une des esquisses d'un des personnages du jeu.



MODÉLISATION 3D

Modélisation des premiers décors

Pierre-Louis se charge de modéliser les décors grâce à Blender.
Les décors se décomposent en de nombreux objets différents.
Voici deux exemples d'objets déjà modélisés.

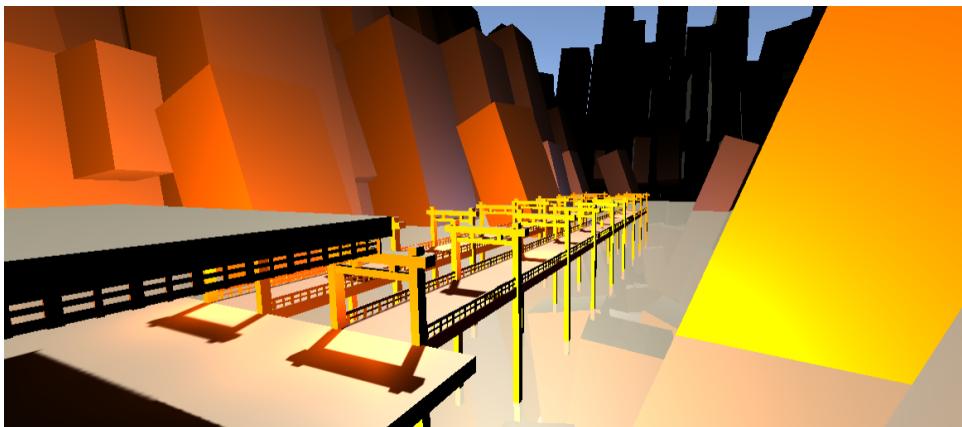
Le premier est une barrière.



Le deuxième est une porte torii japonaise.



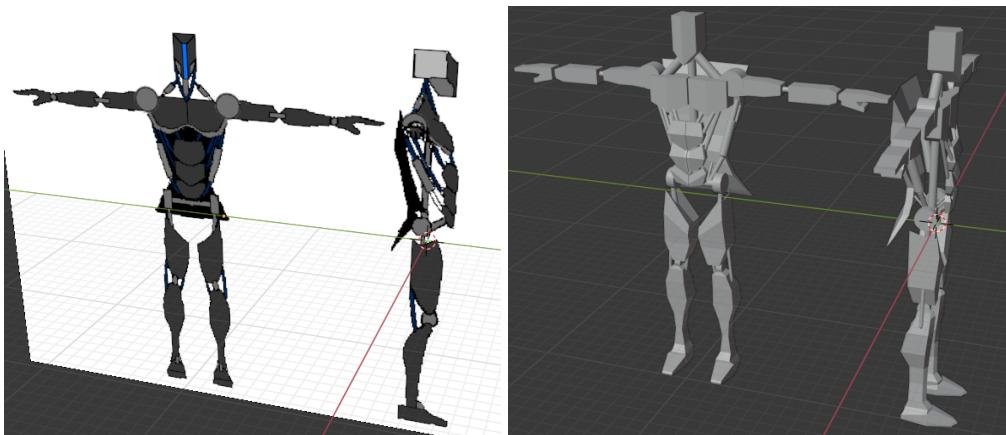
Voici un morceau d'essai de map.



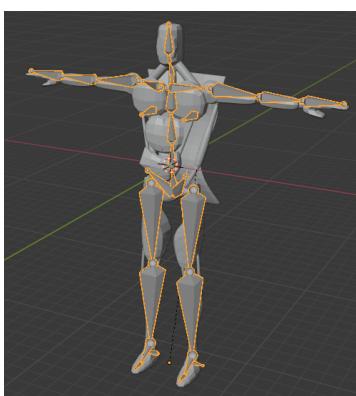
On peut voir qu'elle est composée de plusieurs objets, les portes, les ponts et l'environnement.

Modélisation du personnage

Une fois que Pierre-Louis a terminé de réaliser le modèle 2D du personnage en pose T (la pose en T du personnage favorise la mise en place du squelette 3D). Paul a pu l'incorporer dans le logiciel de modélisation 3D Blender afin d'en réaliser son modèle.



Après une première réalisation dans un format très géométrique et carré. On lui a rajouté des “modificateurs” afin de le rendre beaucoup réaliste. Enfin nous lui ajoutons un squelette, ce qui va permettre de générer des animations fluides à notre personnage.



Ici nous voyons le squelette (en orange) qui est superposé aux articulations et aux membres du robot.

ANIMATIONS

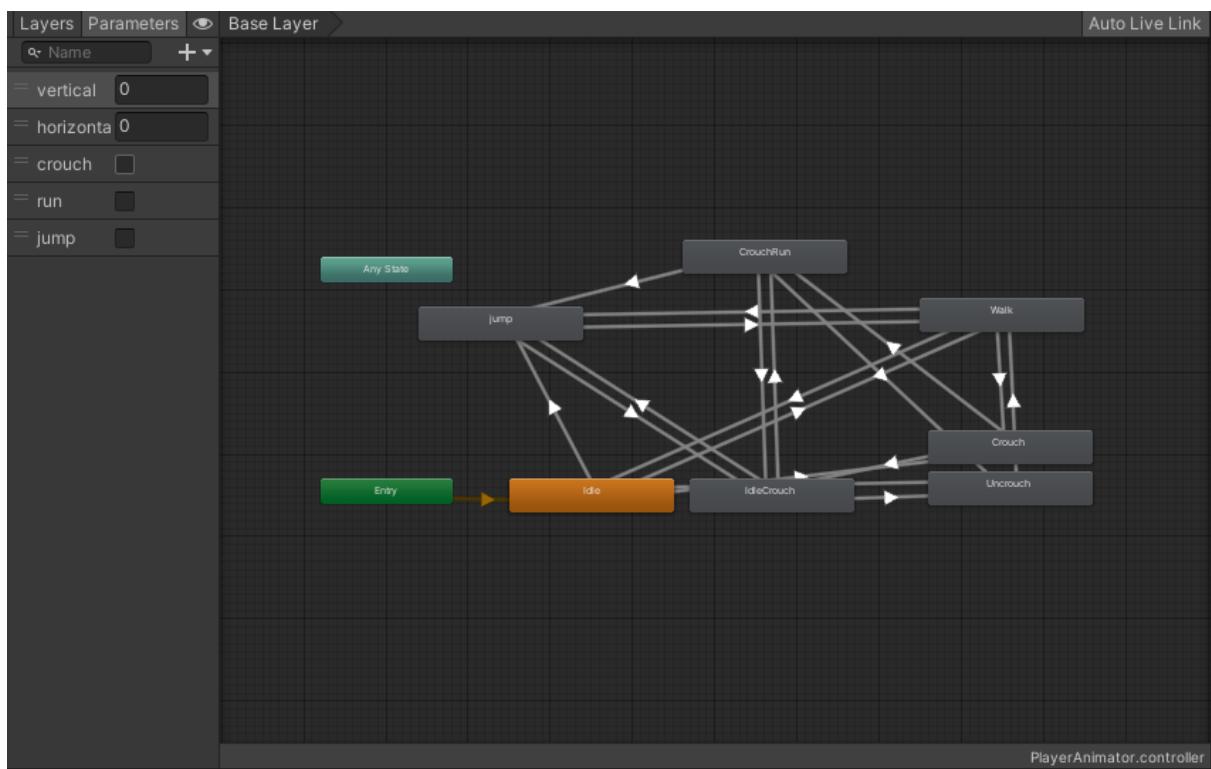
Les animations étant très longues a réalisé nous avons décidé d'utiliser un outil parfaitement adapté à notre situation : Mixamo. C'est un site internet qui permet gratuitement d'obtenir des centaines d'animations pré-faites qui s'adaptent à notre modèle 3D et son squelette. Nous y avons donc récupéré une dizaine d'animations afin de rendre "vivant" notre robot.

Gestion des animations dans Unity

La gestion des animations dans Path Maker se fait de la manière suivante :

- On a importé toutes les animations dans le projet ainsi que le modèle 3D de notre robot.
- On ajoute le "component Animator" à notre personnage et on crée un "PlayerAnimator" qui va permettre de gérer les conditions et l'ordre dans lesquelles se lancent les animations.

Il est composé de 2 parties, une première avec les variables que l'on va pouvoir modifier dans un script, et une partie où l'on retrouve les animations relier entre elles par des conditions de lancement.



- Enfin on créer un script qui va permettre de modifier les différentes variables en fonction des actions du joueur. On fixe le script sur le personnage.

```

public Animator anim;

private bool run = false;
private bool crouch = false;
private bool jump = false;
private float Frun;

```

En premier, on récupère le “PlayerAnimator”.

On crée ensuite des variables afin de faciliter la modification des variables créées dans le “PlayerAnimator”.

Le corps du script :

```

void Update()
{
    if(Input.GetKeyDown(KeyCode.LeftShift)){
        run = !run;
    }
    if(Input.GetKeyDown(KeyCode.LeftControl)){
        crouch = !crouch;
    }

    jump = Input.GetKeyDown("space");

    anim.SetBool("jump", jump);
    anim.SetBool("crouch", crouch);
    anim.SetBool("run", run);

    Frun = Input.GetAxis("Vertical") * 2;

    if(run){
        anim.SetFloat("vertical", Frun);
    }
    else{
        anim.SetFloat("vertical", Input.GetAxis("Vertical"));
    }

    anim.SetFloat("horizontal", Input.GetAxis("Horizontal"));
}

```

Globalement on détecte si le joueur appuie sur certaines touches afin de modifier les variables. Par exemple pour les déplacements sur l'axe vertical Z,S, l'axe horizontal Q,D ou la barre espace pour sauter. Enfin on injecte nos valeurs à l'aide de la méthode “anim.SetFloat ou anim.SetBool”.

SITE INTERNET

Le site internet a été réalisé par Paul en HTML, CSS et JS à l'aide à l'aide du logiciel Bootstrap. Il est actuellement hébergé sur Netlify, voici le lien : <https://pathmaker-ygreg.netlify.app/>

La page d'accueil possède une rapide présentation du jeu ainsi que la présentation de l'équipe. Chaque membre possède sa propre page avec sa présentation.

On retrouve également un lien vers la page de présentation du projet, avec sa nature et ses origines.

Dans la dernière partie il y a les liens vers le rapport de la première soutenance et le cahier des charges.

CONCLUSION

En conclusion de ce rapport, on rappelle ce qui doit être fait pour la prochaine soutenance. Notre groupe étant dans les temps, ce planning n'a pas changé depuis celui rendu dans le cahier des charges.

	Soutenance 1	Soutenance 2	Soutenance 3
Gameplay	33%	66%	100%
User Interface	0%	50%	100%
Modélisation	33%	66%	100%
Texture	33%	66%	100%
Audio	0%	50%	100%
IA	0%	50%	100%
Mode multijoueur	60%	90%	100%
Gestion de compte	40%	80%	100%
Site Internet	42%	84%	100%
Marketing	0%	0%	100%