

Rapport de première Soutenance

History

Elsa FRANCOIS (*Leader*)
Quentin ROBERT
Lorenzo TAALBA
Agl   TOURNOIS

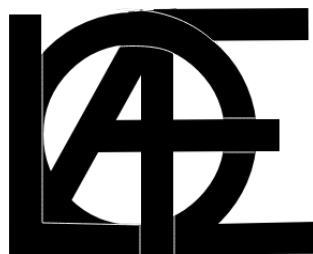


Table des matières

1	Introduction :	3
1.1	Rappel	3
1.2	Tableaux	3
2	Reseau : –Quentin–	5
2.1	Photon	5
2.2	Implémentation	5
2.3	Site	6
2.3.1	Repl.it	6
2.3.2	L'avancement	6
3	Caractère ou chara-design : –Aglaé–	7
3.1	Tableaux de l'avancée	7
3.2	Logiciel et Utilité	7
4	Map / Carte : –Elsa–	8
4.1	Explication	8
4.2	Piskel	8
4.3	Avancée	8
4.4	Implémentation	9
5	Histoire : –Elsa–	10
5.1	Décision générale	10
5.2	Recherche et détails	10
5.3	Rappel :	11
6	Gameplay : –Lorenzo-Quentin–	12
6.1	Génération Des Personnages : –Quentin–	12
6.2	Déplacement : –Lorenzo–	12
6.3	Tetris :	13
7	Combats : –Lorenzo-Quentin–	17
7.1	Combat en lui-même : –Lorenzo–	17
7.2	Colliders : –Lorenzo–	19
8	IA : –Lorenzo-Quentin–	20
8.1	Pong : –Lorenzo–	20
8.2	Street Fighter, Boss : –Quentin–	20
9	UI : –Quentin–	22
9.1	barre de vie :	22
9.2	Paramètres :	22
10	Conclusion	23



1 Introduction :

1.1 Rappel

History est un jeu avec une histoire simple à comprendre, vous contrôlez un robot qui aide un dragon à rejoindre son ère. Pour cela vous aurez le choix entre, solo ou aidé d'un ami. En solo vous contrôlerez les deux, il vous suffira de changer de l'un à l'autre, alors qu'en duo vous contrôlerez chacun un personnage.

L'univers du jeu changera tout au long de l'histoire, elle augmentera petit à petit en commençant en 2D, 16x16pixels jusqu'en 128x128pixels pour passer ensuite sur de la 2.5D et finir sur de la 3D. Vous acquerrez aussi des compétences qui vous seront utiles pour la suite.

Pour tout vous expliquer sans rien oublier commençons, faisons la description de notre avancée dans l'ordre de notre tableau d'avancée.

1.2 Tableaux

Répartition	Elsa	Quentin	Lorenzo	Aglaé
Réseau				
Site				
Caractère				
Maps				
Histoire				
Gameplay				
Combat				
IA				
UI				
Sound				

■ Responsable ■ Suppléant

Attendu \Rightarrow Fait	Attendu	Fait
Reseau	60%	\Rightarrow 55%
Site	00%	\Rightarrow 05%
chara-design	30%	\Rightarrow 30%
Art carte	20%	\Rightarrow 20%
Histoire	70%	\Rightarrow 90%
Gameplay	40%	\Rightarrow 40%
Combat	00%	\Rightarrow 30%
AI	00%	\Rightarrow 10-15%
UI-Interfaces	40%	\Rightarrow 40%
Sound	00%	\Rightarrow 00%



2 Réseau : –Quentin–

2.1 Photon

Pour le réseau, nos ACDC nous ont conseillé Photon, car il permet de se connecter à un même serveur assez simplement et tout ça gratuitement.

Après essai, nous avons donc pu obtenir un serveur qui nous permet actuellement d'héberger nos différentes scènes, qui sont en soit nos différents niveaux. Quelques-unes de ces scènes ne seront pas des niveaux tels que "Loading", ou "Lobby" qui sont essentiellement présents pour les entre-scènes et pour l'entrée ou la sortie d'un niveau.

2.2 Implémentation

Le réseau a été, pour la plupart, fait par Quentin. Pour cela il y a plusieurs étapes importantes à effectuer pour que les deux personnes voulant jouer soient bien dans la même salle et que les mouvements de chacune soient bien raccordés avec ceux des autres.

Pour cela Photon a deux "packages" en particulier qui aident et simplifient beaucoup certaines étapes de la construction de cette connexion.(Image 1)

```
using UnityEngine.UI;
using Photon.Pun;
```

(Image 1)

Et à l'aide de ces deux packages, de (Image 2) et de certaines commandes, les salles se créent et se relient au serveur. (Image 3) (Image 4)

```
PhotonNetwork.
```

(Image 2)

```
public void CreateRoom()
{
    PhotonNetwork.CreateRoom(createInput.text, new RoomOptions() {MaxPlayers = 2});
}
```

(Image 3)

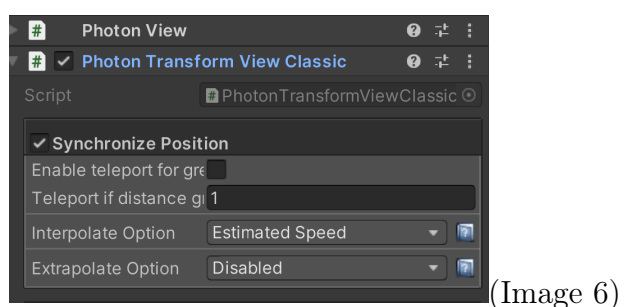
```
public void JoinRoom()
{
    PhotonNetwork.JoinRoom(joinInput.text);
}
```

(joinInput.text = nom de la salle)(Image 4)

Après la connexion des deux personnes à la même salle il ne manque que deux choses, séparer les points de vue et synchroniser les positions.

```
//PhotonNetwork.LocalPlayer
if(view.IsMine)
{
    GetMoove(scene);
}
```

(Image 5)



C'est pratiquement tout pour le réseau, il nous manque 2 points à améliorer/créer :



- un meilleur design pour le Lobby.
- Et la partie pour changer de personnage, mais ce n'était pas dans nos objectifs de rendu.

2.3 Site

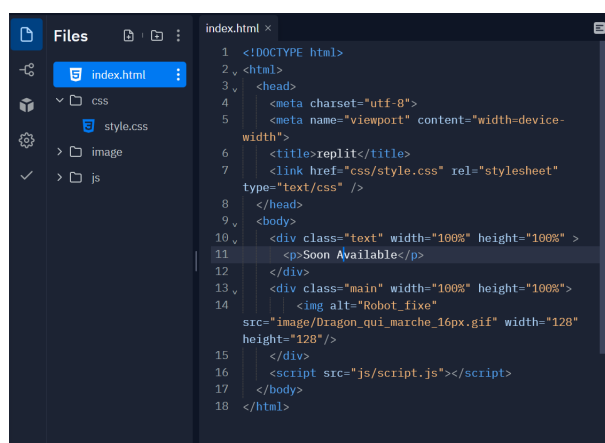
2.3.1 Repl.it

Grâce au logiciel Repl.it, la conception du site à pu légèrement avancer avant cette première soutenance. Elsa a commencé cela sur un logiciel de code colaboratif disponible sur internet. Ce logiciel particulièrement nous permet d'héberger gratuitement notre site.

Il permet également de coder en html et en css ainsi que générer le site de manière instantanée. Cela est pratique lorsque l'on cherche à découvrir le fonctionnement de ce langage.

2.3.2 L'avancement

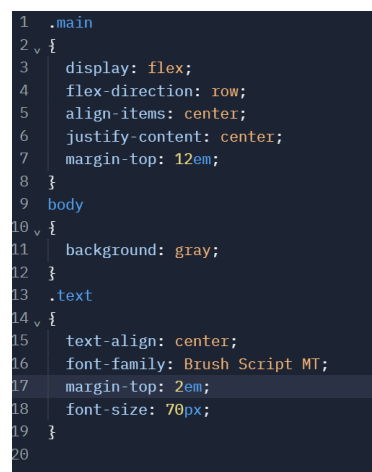
Pour le moment, le site ne contient que deux objets. Un gif de notre personnage patapouf ainsi que un texte indiquant que le site ainsi que le jeux seront bientôt disponibles.



```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <meta name="viewport" content="width=device-
width">
6   <title>replit</title>
7   <link href="css/style.css" rel="stylesheet"
type="text/css" />
8 </head>
9 <body>
10   <div class="text" width="100%" height="100%" >
11     <p>Soon Available</p>
12   </div>
13   <div class="main" width="100%" height="100%">
14     
15   </div>
16   <script src="js/script.js"></script>
17 </body>
18 </html>

```



```

1 .main
2 {
3   display: flex;
4   flex-direction: row;
5   align-items: center;
6   justify-content: center;
7   margin-top: 12em;
8 }
9 body
10 {
11   background: gray;
12 }
13 .text
14 {
15   text-align: center;
16   font-family: Brush Script MT;
17   margin-top: 2em;
18   font-size: 70px;
19 }
20

```

Il y a également un fichier en css qui nous permet de découvrir quelques fonctionnalités du css : la taille d'écriture ainsi que la police, comment centrer une division, la création de marge et gérer un background.



3 Caractère ou chara-design : –Aglàé–

Pour la conception de nos personnages, des éventuels PNJ, d'écran de chargement et ennemi, c'est Aglaé et Elsa qui s'en occupent.

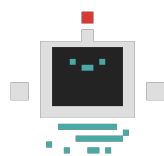
L'avancée actuelle a été faite par Aglaé, il s'agit de la conception de nos héros.

3.1 Tableaux de l'avancée

Pour l'instant elle a créé :

Graphisme / Qualité	Robot	Dragon	En plus
16x16 pixels	Debout // Marche	De face // Debout // Marche	Oeuf : Roule
32x32 pixels	De dos // Debout // Marche	Debout	
64x64 pixels			
128x128 pixels			

■ Pas Commencé



Robot

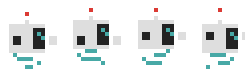


Oeuf

De plus, une icône, à mettre sur le bouton "multi-joueur", a été réalisée.

3.2 Logiciel et Utilité

Pour cela Aglaé a utilisé un logiciel : Piskel, cela lui permet de designer et dessiner ses personnages comme elle le souhaite et de les superposer pour visualiser les mouvements.



4 Map / Carte : –Elsa–

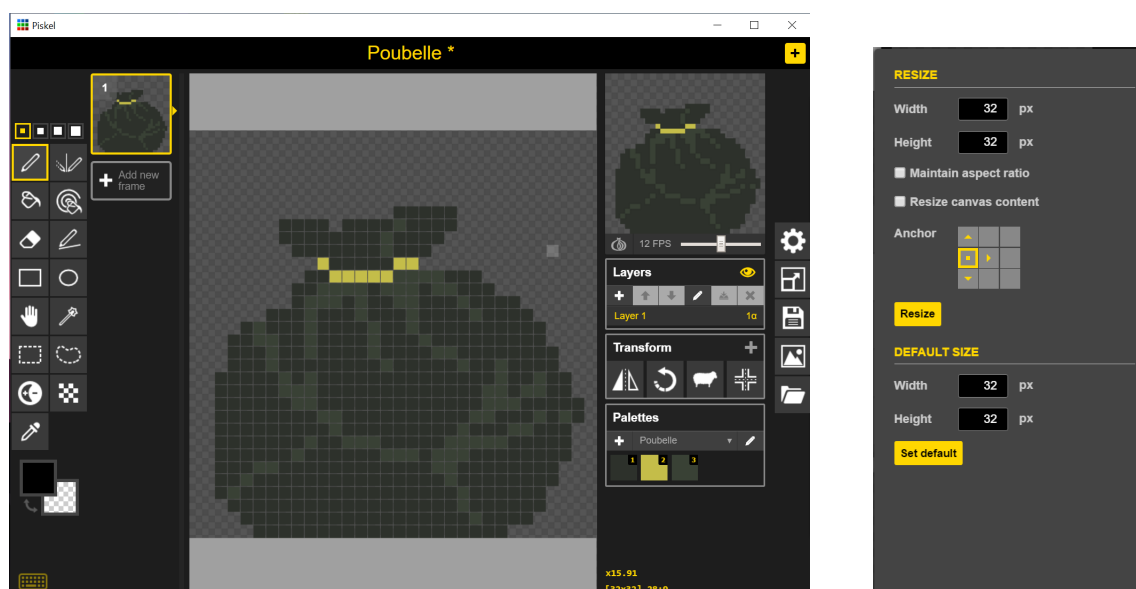
4.1 Explication

Il y aura deux sortes de dites "map", il y aura celle sur laquelle on se déplace, et celle qui arrivera un peu plus tard dans l'histoire et qui permettra au joueur d'avoir une idée de son emplacement dans le monde, comme une carte.

Pour la map sur laquelle on se déplace, tel que le laboratoire ou les ruelles dans un mario, c'est aussi Aglaé et Elsa qui s'en chargent. Mais cette fois, c'est Elsa qui s'en est occupé.

4.2 Piskel

La première étape à été, avec l'aide d'aglaé, d'apprendre à utiliser le logiciel piskel. En modifiant quelques réglages, il est possible d'adpter le nombre de pixel de notre objet. Pour le style mario, nous nous sommes mis d'accords sur des objets en 32x32px.



4.3 Avancée

Grâce a google maps, et le mode street view, il a été possible de créer plusieurs croquis représentant des objet courants dans une rue. Tels que des voitures, des bouches de métro, ou encore un simple lampadaire. Parmi tous ces croquis, peu ont été implémentées dans le jeu car pas encore numérisées.



Allant d'un simple lampadaire, déjà fait sur Piskel, au kiosque, pour le moment sous forme d'un simple croquis dessiné.



Toute notre map sera faite à l'aide d'une "tilemap". Elle nous permet d'ajouter des éléments "cell" par "cell" qui, mis bout à bout, construiront petit à petit l'entièreté de notre map.

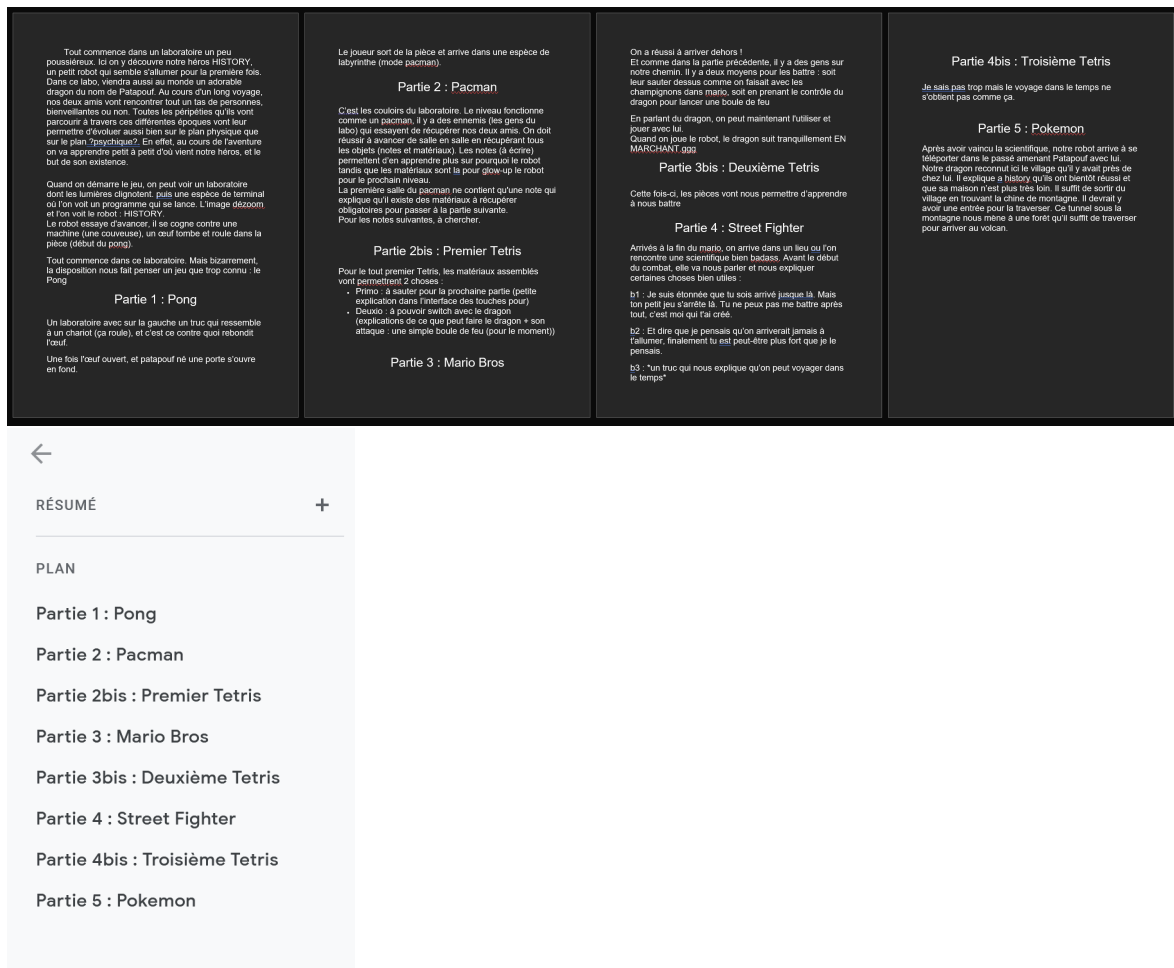
4.4 Implémentation

Son avantage est qu'elle est simple d'utilisation et qu'elle permet d'intégrer facilement les images précédemment créées. Il faut juste un peu d'organisation pour le rangement de tous les "tiles".

5 Histoire : –Elsa–

5.1 Décision générale

En globalité l'histoire a été créée par tous et on s'est tous mis d'accord sur quelque chose qui nous convenait. Mais pour ce qui est des détails de chacun des passages, des transitions et des éventuels évènements, c'est aussi Elsa qui s'en occupe. Elle a fait un document de quatre pages avec une description de l'histoire et des péripéties et des petits détails à ne pas oublier.



5.2 Recherche et détails

Pour résumer l'histoire globale :

- Tout commence dans un laboratoire où notre robot, History, fait la rencontre du bébé dragon, Patapouf.
- Après être sortis du laboratoire, nos deux héros courent dans les rues pour échapper aux scientifiques.
- Arrive le moment d'affronter une scientifique qui l'a créé et qui lui apprend qu'il peut se téléporter dans le temps.
- Après ça, ils se retrouvent dans une nouvelle ère, la préhistoire, aux portes de la même ville mais des décennies plus tôt.
- Ici nos deux personnages pourront se déplacer dans un monde semi-ouvert (avec la carte



précédemment évoquée).

-Ici vous devrez surmonter plusieurs obstacles et résoudre plusieurs énigmes pour vous frayer un chemin dans un monde hostile.

-Ainsi, nos deux héros parcourront toutes sortes de paysages jusqu'à arriver à la tanière des parents du dragon, Le Volcan. Et c'est ici que la 3D arrive avec une animation de fin.

5.3 Rappel :

Ce qu'il faut tout aussi savoir c'est que le jeu reprendra des anciens jeux. Nous en recoderons certains, d'autres où l'on reprendra l'idée et d'autres encore où nous ne ferons que des références. Si on reprend les étapes précédemment dites, on associera :

Début \implies référence au Pong.

Laboratoire \implies couloir et gameplay à la Pac-Man.

Sortie dans les rues \implies style à la Mario Bros.

Combats avec le scientifique \implies référence au Street Fighter.

Retour dans le temps \implies map à la Pokemon.

Puis quelques mini-jeux où, pour les réussir il faudra compléter un Tetris complètement recodé.



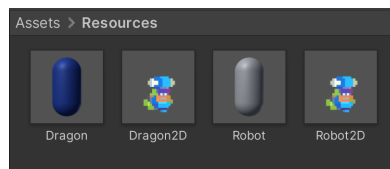
6 Gameplay : –Lorenzo-Quentin–

6.1 Génération Des Personnages : –Quentin–

Pour ce qui est du gameplay, Photon va encore être nécessaire pour l'apparition de nos personnages. En effet bien que la scène soit créée et implémentée, il faut que nos héros puissent y être "spawnés" (Image 8) et que l'on puisse s'y déplacer.

```
if (PhotonNetwork.CountOfPlayersInRooms == 0)
{
    Vector3 spawn = new Vector3(0, 16, 37);
    PhotonNetwork.Instantiate(RobPlayer.name, spawn, Quaternion.identity);
}
```

Il faut donc des capsules, représentant nos héros, que nous mettons dans le dossier "Ressources" pour qu'ils puissent être détectés par Photon. Nous les créons en tant que "GameObject" pour pouvoir les instancier dans notre "SpawnPlayer" qui va ensuite pouvoir les faire apparaître sur notre dite scène.



6.2 Déplacement : –Lorenzo–

Après l'apparition de nos capsules, qui seront donc en 2D, il faut pouvoir les déplacer sur notre terrain. Pour cela nous avons organisé le code de la manière suivante :

```
1 reference
public void GetMoove(int type)
{
    if (type == 1)
    {
        this.x = Input.GetAxis("Horizontal");
        this.y = Input.GetAxis("Vertical");
    }
    else if (type == 2)
    {
        this.x = Input.GetAxis("Horizontal");
        this.y = Input.GetAxis("Jump");
    }
    else if (type == 3)
    {
        this.x = Input.GetAxis("Horizontal");
        this.y = Input.GetAxis("Vertical");
        this.z = Input.GetAxis("Jump");
    }
}
```

```
public void Moove(int type)
{
    if (type == 1)
    {
        transform.Translate(Vector3.right * x * forcemoove);
        transform.Translate(Vector3.up * y * forcemoove);
    }
    else if (type == 2)
    {
        transform.Translate(Vector3.right * x * forcemoove);
        if (onfloor)
        {
            player.AddForce(Vector3.up * y * forcejump, ForceMode2D.Impulse);
        }
    }
    else if (type == 3)
    {
        transform.Translate(Vector3.right * x * forcemoove);
        transform.Translate(Vector3.forward * y * forcemoove);
        if (onfloor)
        {
            player.AddForce(Vector3.up * z * forcejump, ForceMode2D.Impulse);
        }
    }
}
```

Les déplacements vus ci-dessus sont appelés dans la fonction "Update" ce qui permet à "Moove" et "GetMoove" de tout le temps être à jour pour les mouvements.

Ce système marche car il y a aussi un script, héritant du script précédent, attaché à chacun des joueurs comportant un "Dictionary" avec le nom des scènes associé à un nombre.

```
public class Robot : Player
{
    0 references
    void Start()
    {
        scene_to_ttypemoove = new Dictionary<string, int>()
        {
            {"Game" , 1},
            {"Pong" , 1},
            {"Pacman" , 1},
            {"Mario" , 2},
            {"Street_Fighter" , 2},
            {"Pokemon" , 3}
        };
        base.Init();
    }
}
```

Cela permet donc de donner à "Moove" et "GetMoove" les informations nécessaires pour savoir quels types de déplacement sont autorisé aux joueurs, dans quelle scène.

Détails :

-Dans l'image 8, celle pour le "spawn" de la capsule, une condition est présente pour pouvoir différencier et faire "spawner" les deux joueurs avec deux différents personnages, et cela permet aussi de ne pas avoir plus de 2 personnes dans un jeu.

-Le Moove, qui va faire déplacer le jouer, est appelé dans la fonction "FixedUpdate" qui permet de faire une update 60 fois par seconde, donc à intervalle de temps régulier.

-Pour éviter de sauter à l'infini, une variable "onfloor" est ajoutée qui vérifie à l'aide des "colliders" ajoutés à notre personnage s'il touche le sol ou pas.

6.3 Tetris :

C'est dans cette phase que le joueur pourra améliorer son robot ou apprendre de nouvelles capacités à son dragon, elle arrivera entre chaque level. Dans ce jeu, il vous faudra compléter au maximum un plateau grâce à des formes, semblables à celles trouvables dans un Tetris, qui tombent. On peut les bouger de droite à gauche et les tourner. Quand il n'est plus possible d'en ajouter davantage et qu'on touche le "plafond", le nombre de cube déterminera si on accédera à l'amélioration ou si on réessaye le mini-jeu.

L'implémentation du Tetris se fait sur deux parties. Premièrement, une partie non visible sera représentée par deux nouvelles classes : un TetrisBoard représentant le plateau de jeu et un TetrisShape étant la forme qui tombe. Et deuxièmement, une partie visuelle sur Unity. Le Tetris est composé de 3 scripts en tout, Tetris, Tetrisboard et Tetrisshape.

TetrisBoard :

D'abord, commençons par détailler le plateau du tetris. Un plateau de Tetris est composé d'une largeur, d'une hauteur et d'une taille (le nombre de cases du plateau) ; d'une position d'où apparaîtront les formes ; d'un gameobject permettant de fabriquer les murs sur Unity et d'une matrice de booléens (false si la place est vide, true s'il y a déjà une forme ou un mur).

Il y a deux manières de fabriquer un plateau de jeu de tetris. Soit en spécifiant sa taille (width et height) et un gameobject utilisé comme une préfab pour construire le plateau dans Unity. Soit en donnant un autre Tetrisboard (plateau de jeu) et de le dupliquer. La première manière de construire un plateau utilise deux fonctions, une fonction (field) qui permet de remplir la matrice de false, sauf sur la ligne du bas et sur les côté qui seront en true, elle même



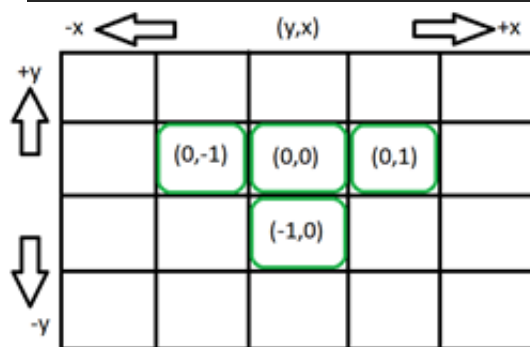
utilisant une fonction `display_wall` qui permet de créer chaque mur sur Unity du plateau grâce à la préfab. Pour la deuxième méthode, elle ne fait que répliquer le tableau donné en paramètre.

Tetrisboard n'a qu'une seule méthode pour l'instant, `Copy`, celle-ci étant utile que pour la deuxième manière de construire le tableau, elle remplace la matrice du plateau par celle entrée en paramètre.

Tetrisshape :

Tetrisshape est défini par une liste de gameobject (une array qui sera utile pour l'implémentation sur Unity), une liste de couple d'entier (une array de position (y,x) avec y et x des int) qui représente la position de chaque carrés de la forme par rapport à un carré de référence (x et y = 0). Dans cet exemple, on représente une forme de T.

```
static private (int, int)[] T = {(0, 0), (0, -1), (0, 1), (-1, 0)};
```



Et enfin une liste de ces listes de positions pour enregistrer toutes les formes possibles.

TetrisShape possède aussi deux manières d'être créé. Premièrement, on peut lui associer un gameobject, utilisé ici comme préfab, puis un Tetrisboard pour connaître la position initiale ainsi que la taille de la matrice. Il va d'abord initialiser la forme qu'aura le tetrisshape grâce à la fonction `rand_char` qui va prendre aléatoirement une des forme dans la liste des formes possibles grâce à `System.Random`.

Ensuite il va l'implémenter dans Unity grâce à la fonction `create_cube`. Cette fonction va prendre la liste des positions de chaque carré, un gameobject et un tetrisboard. Il a pour but de créer pour chaque position des carrés, un gameobject sur unity, qui sera à la position calculée grâce à `mat_to_vact_pos`. Celui-ci transforme l'emplacement d'un objet dans la matrice en un emplacement dans Unity. L'autre méthode est aussi utilisée pour dupliquer un tetrisshape, on lui rentre en argument un tetrisshape. Il créera alors un nouveau tetrisshape qui aura la même forme.

TetrisShape a une méthode `Copy` qui, comme pour Tetrisboard, permet de remplacer la liste des positions des carrés par celle du tetrisshape entré en paramètre. Il a une fonction `rotate_of` qui transforme la forme en la tournant de 90° dans le sens horaire. Et une dernière fonction, `display`, qui permet d'afficher le tetrisshape dans Unity. Il va convertir chaque position grâce à `mat_to_vact_pos` et transforme la position de chaque gameobject.

Tetris :

Ce script est le programme principal. C'est dans celui-ci que toutes les fonctions seront appelées. Dans ce programme on aura besoin : De deux préfabs qui serviront pour créer les murs et les



carrés des formes.

D'un entier qui comptera le nombre de carrés tombés, il servira à vérifier si le joueur à gagner ou s'il doit recommencer. D'un Tetrisshape, la forme actuelle qui tombe et que l'on contrôle, et d'un couple d'entier qui est sa position.

D'un Tetrisboard qui sera le plateau de jeu du tetris et d'un booléen qui deviendra false quand la partie terminera. Et enfin, de deux entiers, right et rotation qui sont respectivement, le déplacement (1 pour le déplacer une fois à droite ou -1 pour décaler à gauche) et le nombre de rotation de la forme à effectuer.

Une des fonctions la plus importante est rm_add qui va effacer ou ajouter une forme dans la matrice du plateau à une position donnée et va retourner true ou false selon si la modification à pu se faire ou non. Pour cela il va pour chaque carré de la forme calculer leur place dans la matrice et soit la supprimer en la mettant a false soit l'ajouter en mettant true, si aucun cube ajouter n'est sur un mur ou sur une forme il return true sinon les modifications entrent en collision et donc retourne false.

Il a par la suite 3 fonctions ayant la meme structures et qui se charge de la rotation, des déplacements horizontaux et de faire tomber la forme. Les fonctions vont d'abord dupliquer le plateau de jeu grâce au deuxième constructeur et la forme pour la fonction qui effectue une rotation. Puis elle efface la forme dans le nouveau tableau.

```
//réplique un nouveau tableau
TetrisBoard new_board = new TetrisBoard(game);
TetrisShape new_obj = new TetrisShape(obj);

//supprime l'la forme actuelle
rm_add(obj, add_rm: false, new_board);
```

Ensuite elle effectue la modification (sur la position pour les fonctions de déplacements ou sur la forme sur la fonction de rotation).

```
//tourne la forme
while (rotation > 0)
{
    new_obj.rotate_of();
    rotation--;
}
```

```
//déplace la position
pos = (pos_y , pos_x + moove);
```

Par la suite, si on peut ajouter la forme avec les nouvelles modifications dans le plateau dupliqué, on la copie ensuite dans le plateau originel sinon on laisse le plateau (et la forme pour la fonction de rotation) comme elles étaient avant.

```
//si on peut ajouter la forme tourner alors remplace les originaux
if (rm_add(obj, add_rm: true, new_board))
{
    game.Copy(new_board);
    obj.Copy(new_obj);
}
Destroy(new_board);
Destroy(new_obj);
```

Et enfin on supprime les copies puis on remet les variables des modifications à zéro. La fonction `create_cube` sert à créer une forme sur le plateau à la position initiale. Elle recopie le plateau, et essaye d'ajouter la nouvelle forme. Si c'est possible, on remplace le plateau par la réplique et en retourne un booléen de si l'ajout a pu se faire.

Pour commencer le jeu on initialise le plateau de jeu, la première forme et la position initiale, ensuite on appelle une fonction `loop`.

Dans `loop` on va récolter les modifications à faire grâce à `right` et `rotation` et effectuer les fonctions associées.

Après ces modifications on met à jour sur Unity grâce à `display`, puis on fait tomber le cube, si on ne peut pas le faire tomber plus bas c'est qu'il a touché un cube ou le sol. Alors on ajoute le nombre de cubes tombés et on crée une nouvelle forme. Si on ne peut plus créer de forme c'est que la partie est terminée! On affiche finalement le nombre de cubes tombés, on remet à jour la chute de l'objet et on rappelle `loop` chaque seconde.

```
//déplace la position
pos = (pos_y + 1, pos_x);
```


7 Combats : –Lorenzo-Quentin–

Commençons par les basics puis allons dans les éléments plus compliqués par la suite.

Pour que le combat se passe bien il nous faut un Boss et notre héros, cependant n'étant pas encore designées en 128x128, nous sommes allés chercher des personnages libres de droit. Nous avons donc posé sur notre héros un des scripts précédemment évoqués pour qu'il puisse se mouvoir dans un terrain de test.

A partir d'ici, le jeu se sépare en trois principaux points :

- Le combats lui-même : les attaques et collisions.
- L'IA : Les déplacements du boss et ses différentes phases durant le combat.
- L'UI : la barre de vie de chacun des combattants et leur mise à jour.

7.1 Combat en lui-même : –Lorenzo–

Partons du principe que la barre de vie marche et qu'en cas de dégâts tout est raccord (elle sera expliquée dans la partie UI). Là aussi, on peut décomposer cette partie en trois autres sous-parties :

- La création du projectile/de l'attaque.
- Son déplacement.
- S'il touche/cas d'arrêt.

Pour la conception du projectile en lui-même, sans que rien ne lui soit appliqué (déplacement, collisions), nous avons développé une fonction qui avec sept paramètres permet de le créer.

Arguments	Dégâts	"Effect"	Taille du Projectile	Position Lancement	Temps Restant	"Type"	Arg
Exemple de Projectile	10	" " (0 effect)	2	position du joueur	100	"project"	new Args()
Exemple de Coups	15	" " (0 effect)	10	position du joueur	100	"around"	new Arg()

```

2 references
void atk(int deg, string effect, int size, Vector3 pos, int time, string type ,Args arg)
{
    GameObject hit = Instantiate(this.hit, pos, Quaternion.identity);
    hit.transform.localScale = new Vector3(size, size, size);

    hit.GetComponent<SF_Atk>().info = new Tuple<int,string>(deg, effect);
    hit.GetComponent<SF_Atk>().time = time;
    hit.GetComponent<SF_Atk>().arg = arg;
    hit.GetComponent<SF_Atk>().type = type;

    Destroy(arg);
}

```



A la dernière colonne du tableau, il y a cette variable "Arg()" qui est une nouvelle classe. Elle permet d'éviter d'"overrider" la fonction "atk" en entier mais de ne changer que cette petite partie.

```
1 reference
public Args(Vector3 dir, float speed, float gravity)
{
    this.arg2 = dir;
    this.arg3 = speed;
    this.arg4 = gravity;
}
```

```
1 reference
public Args(GameObject player)
{
    this.arg1 = player;
}
```

Comme vu ci-dessus selon les arguments qu'on lui donne, elle s'en sert soit pour donner une direction, vitesse et gravité au projectile, soit pour donner le GameObject parent de la fonction (Dépend si c'est un "project" ou un "around").

L'avantage de cette méthode, bien qu'un peu poussée, nous permet d'avoir une très bonne organisation au niveau des différentes attaques. S'il y a besoin d'ajouter une quelconque attaque, il n'y a que quelques lignes à ajouter pour qu'elle soit implémentée. "Arg()" permet donc ici ne pas avoir à tout réécrire en cas de nouveau type d'attaque mais simplement quelques éléments.

Bien l'attaque a été créée avec tous ses paramètres. Passons donc à la deuxième partie qui concerne le déplacement des projectiles. A chaque lancer, un nouveau "GameObject" est créé avec toutes les caractéristiques qui lui sont associées grâce à la fonction précédente.

Dans la fonction « FixedUpdate », deux paramètres sont regardés. Le premier est le type de projectile et le deuxième est si le temps est à 0 ou pas pour sa destruction. Si le projectile est un « around » rien à changer l'attaque se fait autour du joueur :

```
1 reference
private void around()
{
    GameObject player = this.arg.to_around_arg();
    this.transform.position = player.transform.position;
    this.time--;
}
```

En revanche si le type est un "project" plusieurs étapes se font : Tout d'abord les éléments de "Arg()" sont récupérés, ce qui permet de mettre à jour la position de l'Object et de son temps restant.

```
1 reference
private void project()
{
    (Vector3,float,float) arg = this.arg.to_project_arg();
    Vector3 dir = arg.Item1;
    float speed = arg.Item2;
    float gravity = arg.Item3;
    dir.y -= gravity;

    if (this.transform.position.x < 50 && this.transform.position.x > -50 &&
        this.transform.position.y < 50)
    {
        this.transform.Translate( dir * speed * Time.deltaTime);
    }
    this.time--;
}
```



Détails

Pour que le projectile ait tout le temps une position/direction, une variable "Vecteur3" est créée. Cela permet à la fonction « updir » de donner au projectile une valeur de base et au cas où le joueur effectue un mouvement en même tps, la variable ne sera pas prise en compte et le projectile sera lancé en fonction du mouvement.

```
3 references
private Vector3 dir = new Vector3(1, 0, 0);

1 reference
void updir()
{
    float x = this.GetComponent<Player>().x;
    float y = this.GetComponent<Player>().y;
    if (x != 0)
        dir.x = x;
    dir.y = y;
}
```

7.2 Colliders : —Lorenzo—

Et pour finir, une petite partie sur les colliders. Ils nous permettent ici de pouvoir détecter quand un projectile rentre dans son champ. Mettre le collider du Boss en "is Trigger", ce qui permet à la fonction "OnTriggerEnter2D" de s'activer, récupérer les éléments de l'objet et ensuite, selon son effet et ses dégâts le boss perdra de la vie.



8 IA : –Lorenzo-Quentin–

Bien que dans notre précédent tableau d'avancée nous avons mis 0% sur la progression de l'IA, nous avons finalement commencé quelques petits projets et recherches sur le sujet. D'où la montée en pourcentage de 0 à 10-15 ne sachant pas vraiment quelle quantité de travail cela pouvait représenter.

Bien que n'étant pas de l'IA très poussée comme nous prévoyons de le faire, nos deux projets sont des éléments essentiels au développement de notre histoire. L'un compose, en quelque sorte, le commencement de notre aventure : Le Pong. Et l'autre est l'épreuve pour le passage dans le temps antérieur, le combat avec la scientifique : Le Street Fighter.

8.1 Pong : –Lorenzo–

Pour que le but du Pong soit mieux compris, le début de l'histoire commencera, en quelque sorte, par l'œuf présenté et réalisé par Aglaé dans chara-design, qui rebondira d'un côté à l'autre d'une pièce.

Le robot déjà présent sur la scène devra le toucher pour commencer le jeu.

C'est donc dans cette optique que nous avons créé de deux petites plateformes, pas encore modélisées, qui se déplacent en fonction de l'œuf pour qu'il puisse rebondir dessus.

La création de fonctions permettant à une plateforme de se mouvoir d'elle-même a permis par la suite de pouvoir, en fonction de l'avancée de la balle/œuf, générer un code qui faisait se déplacer le plateau au bon endroit pour la réceptionner de l'objet.

Il ne manquait plus qu'à ce que le programme s'effectue lorsque l'œuf se déplace dans la direction de l'obstacle, ce qui permet d'avoir une désynchronisation des plateaux.

Détails :

Un collider a été ajouté à l'œuf lui permettant de capter les collisions rencontrées et grâce à un dernier programme, inverser le x lors de l'impact avec les côtés et les y lors de l'impact avec le haut et bas de la scène.

8.2 Street Fighter, Boss : –Quentin–

Pour ce qui est de l'IA du boss, il est encore loin d'être complet. Pour l'instant deux évènements lui sont implémentés : la possibilité de se mouvoir pour reculer si un joueur s'avance trop près de lui ; et la deuxième, elle, s'active lorsque sa vie descend en dessous de 50%.

Revenons aux déplacements :

Tout d'abord, tout ce qui va suivre va se situer dans la fonction "FixedUpdate" ou "Start" qui permettra à certaines valeurs d'être initiées.

Pour le déplacement du boss si le joueur est trop près, tout a été géré avec des distances et des "Vecteur3", bien que l'on soit en 2D.

A savoir, avec l'ajout de ".transform.position" à un "GameObject" on récupère sa position sur la scène.



Que le Jeu soit en solo ou en multi-joueurs, une "array" a été créée, avec à l'intérieur des "GameObjects", taggés "Player"(nos héro), et avec le nombre de joueurs présents.

```
//permets de récup le nbr de joueur et une array avec ces gameobjet tagé Player
joueur = GameObject.FindGameObjectsWithTag("Player");
nbrjoueur = joueur.Length;
```

Leur position est ensuite comparée avec celle du boss, dont on récupère les coordonnées de la même manière car lui aussi a un tag : "Ennemi".

A partir d'ici si la comparaison est fausse, il ne manque plus qu'à voir de quel côté le player arrive, pour faire mouvoir le boss dans la bonne direction.

```
//tout le para qui suit sert a:
Vector3 boss = GameObject.FindGameObjectWithTag("Ennemi").transform.position;//récup position boss
if(Vector3.Distance(joueur[0].transform.position, boss) < 10f)//joueur0 de l'array si il s'approche trop
{
    if(joueur[0].transform.position.x < boss.x)//récupère la dif de x => bouhe le boss du bon coté
    {
        Movennemi(mvboss, 1);
    }
    else
    {
        Movennemi(mvboss, (-1));
    }
}
```

Pour finir il ne manque donc plus qu'à lui ajouter les mouvements.

Lorsqu'il descend en dessous des 50% de vie :

Lorsque la phase s'active, le boss s'envole au-dessus de la map, attend quelques secondes au centre puis fait des allers-retours entre les deux côtés du terrain.

Le principe de distance, utilisé ci-dessus, va être réutilisé mais de manière différente. En effet, trois points ont été préalablement placées en hauteur. Ils ont été placés dans une "array" de "Vecteur3" ce qui permet d'y avoir accès facilement et de changer d'un point à un autre juste avec une addition et un modulo.

Maintenant que nos trois points sont initiés il ne reste plus qu'à se déplacer de l'un à l'autre, avec comme condition de changer de destination si l'on s'approche trop d'un point.

```
//decide la direction : cible - position actuel
Vector3 avance = targets.position - transform.position;
transform.Translate(avance.normalized * mvboss * Time.deltaTime, Space.World);

//si arrive près du pts alors on change le sens
if(Vector3.Distance(transform.position, targets.position) < 0.3f)
{
```

Détails :

Le boss a un "Rigidbody2D" ce qui lui permet de se mouvoir et d'être affecté par la gravité. Il est donc primordial de le désactiver pendant cette phase. Cela se fait avec "rb.isKinematic = true" qui désactive la gravité du boss.



9 UI : —Quentin—

Pour finir sur ces longues explications, nous terminerons sur l'UI : « User Interface ».

Elle sera séparée en deux parties :

- la barre de vie de boss.
- le menu de paramètre.

9.1 barre de vie :

Quelque soit la barre de vie, elle sera, la plupart du temps, composée de deux éléments : d'une image de la couleur voulue pour la vie, et superposé par-dessus, les contours de la barre de vie.

Pour que la barre de vie soit reliée au code pour qu'il puisse être modifié lors d'une quelconque attaque, il faut attacher un « Sliders » à l'image. Il sera ensuite accessible depuis les scripts.

A partir de là, dès lors que l'on change la valeur du « Slider » à l'intérieur dans le code, la barre de vie du boss changera elle aussi.

Détails :

C'est donc ici qu'est géré la perte de vie de l'ennemi.

9.2 Paramètres :

Lorsque le joueur appuis sur la touche « échappe », plusieurs boutons apparaissent, « Settings, Return, Quit ». Ils sont tous reliés à un programme différent qui soit : ouvre un nouveau UI pour les settings, soit désactive le UI pour retourner sur le jeu, soit quitte le Jeu.



10 Conclusion

Donc pour conclure, notre projet avance bien. Le plus dur, qui est la compréhension de Unity et comment l'utiliser, est fait. Il ne manque donc plus qu'à créer les salles, leur design, les codes associer et l'implémentation de sons et le développement du site. Bien que cela reste beaucoup, une fois que l'on s'est lancé le reste est plus simple.

