Rapport de Soutenance

Elderberried

Table des matières

1 Description et récit de la réalisation

1.1 L'architecture

1.1.1 Utilité de l'architecture du projet

La programmation, tout le monde le sait, ça peut vite devenir désordonné. L'utilité des techniques lié à l'architecture dans le code tel que la hiérarchie des dossiers, des classes et la réutilisation de composants et de fonctions est donc crutial dans la réalisation de notre jeu.
Plus complexe encore, le type de **party game** exige une attention toutes particulière à la **modularité** car il rapproche des jeux qui ont des fonctionnements très différents.

1.1.2 Avant la première soutenance

Avant la première soutenance il était important de faire la base du projet ce qui rendait la conception d'architecture imparfaite à cause du manque d'expérience dans le moteur de jeu.

1.2 Evolution de l'achitecture

Dès notre choix de type de jeu, c'est-à-dire un Party Game avec des mini-jeux, nous nous sommes tout de suite dit qu'il nous fallait un système d'implémentation facile pour un ajout rapide et efficace de jeux.
Nous y avions mis beaucoup d'effort et avions donc fait en sorte que toutes les informations soient stocké dans chacun des joueurs, et que les joueurs changerai d'apparence et de mouvement selon le mini-jeu, en désactivant tous les composants des autre mini-jeux au passage.

Comme vous pouvez vous en douter, ce système nécessitait de garder en permanence les informations de tous les mini-jeux sur le joueur. Donc Thomas.H, le membre résponsable de l'architecture, a décidé de changer ce système : le joueur deviendra un manager de ces composants, il les fera s'activer et ce désactiver, mais ne changera plus d'apparence, et ne subira plus d'action spécifique au mini jeux.

Mais cela n'a pas réglé le problème principale, que les données soient gardée dans le player, même quand on ne se situe pas dans les mini-jeux spécifique.

Nous en arrivons donc à l'architecture qui a été utilisé jusqu'à aujourd'hui, utilisant les interfaces IPlayer et IGameManager, qui sera présentée ci-dessous.

1.2.1 Base

Le projet est modulé selon une classe **'NBPlayerBase'** qui fait partit de la "prefab" du joueur, celle ci est également raccorder à un gestionnaire d'input présent dans la class 'InputManager'.
Tout composant étant nécessaire pour changer l'aspect et le fonctionnement du joueur seront donc directement relié à sa prefab pour éviter les complications.

Cette architecture à survecu et est toujours la base de notre projet avec deux trois composant en plus. Un schéma représentant ces compositions est présent en figure :

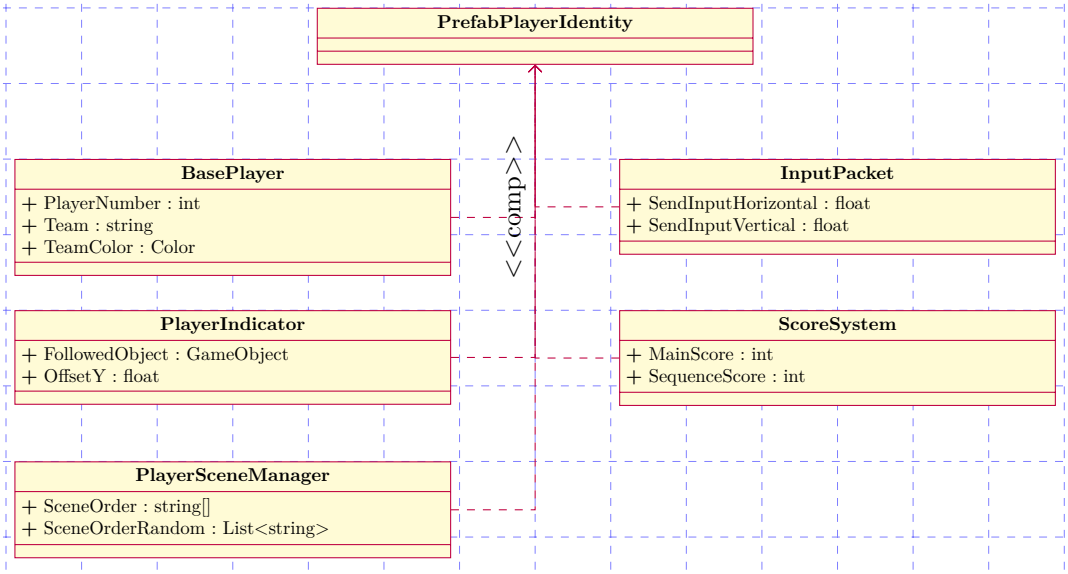


FIGURE 1 – Base Components

1.2.2 Mouvements

Il n'a finalement jamais été réellement utile de faire les mouvements en interface car elle constituent un très petit temps de programmation et que quelque jeu comme **TankMiniGame** suivent une logique extrêmement compliqué en terme de mouvement ce qui nous aurais bloquer car nous aurions été obligé de passer par une interface réstreinte.
Voici cependant l'idée initiale :

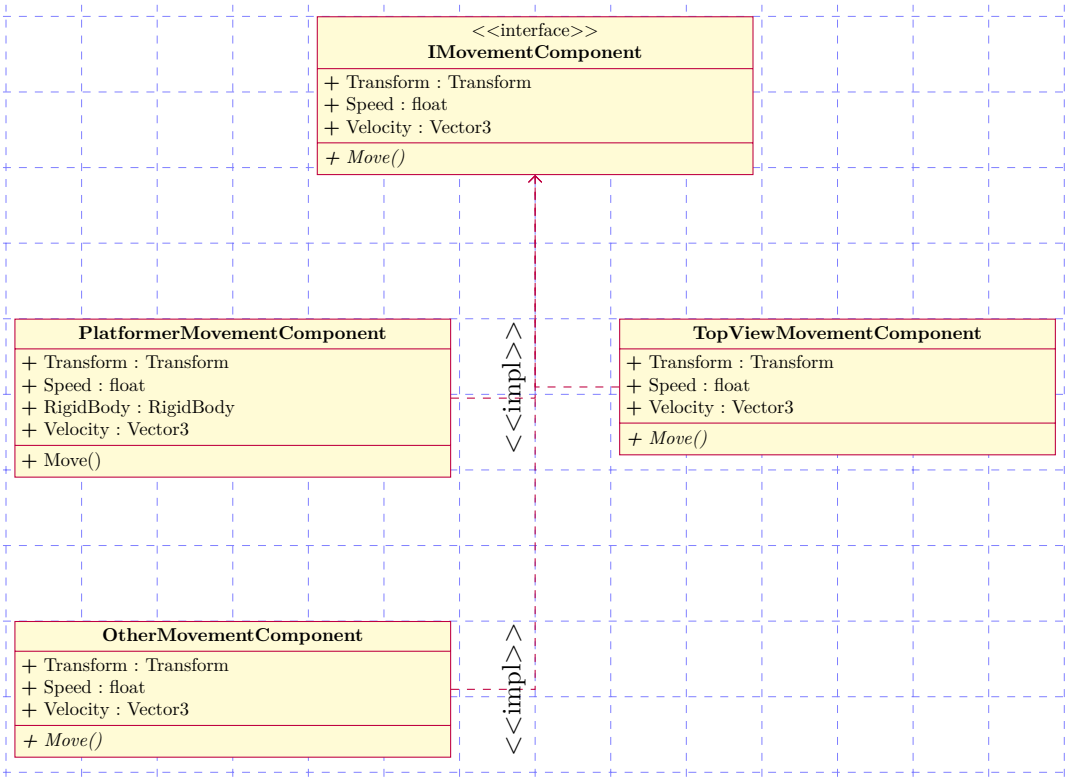


FIGURE 2 – Movement Components

1.3 Entités

Voici un exemple d'implémentation de script de base pour chaque entités qui ne sera pas utilisé et remplacer par des interface ou rien du tout (le principe s'opposé en réalité à l'architecture en composant même si elle est peu présente) :

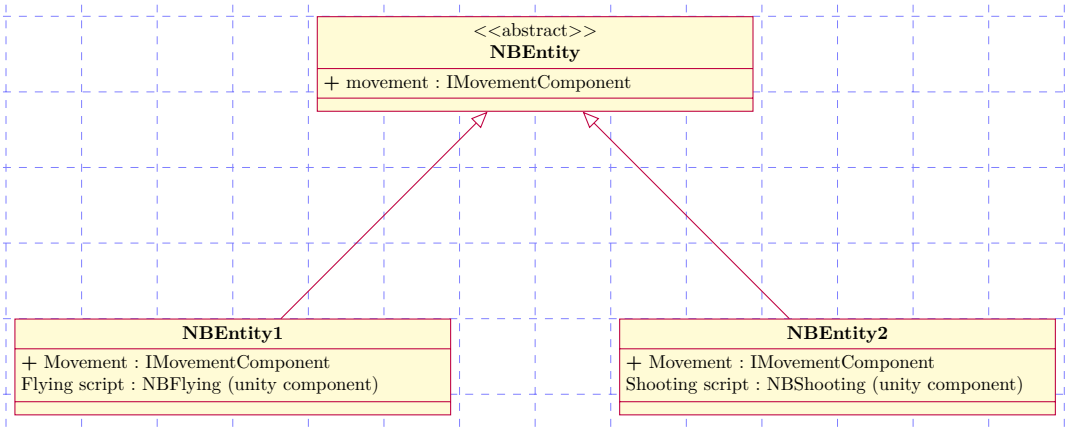


FIGURE 3 – Entity inhéritance

1.3.1 Architecture en composant

Une de nos aspiration initiale lors de ce projet était de réaliser une architecture fais de petits composants pouvant facilement se poser sur n'importe quelle entité ou joueur, l'expérience nous a cependant appris que ça n'était pas forcément intéressant en raison de la différence comportemental de nos jeux il était donc plus facile de réaliser des scripts contenant tous ce qu'il faut pour faire fonctionner un joueur.
Si jamais il y a besoin de réutilisation nous pouvons simplement demander de l'expertise ou des bout de code à un autre programmeur.
Voici cependant un exemple d'architecture en composants venant du mini jeu **PushPush**.

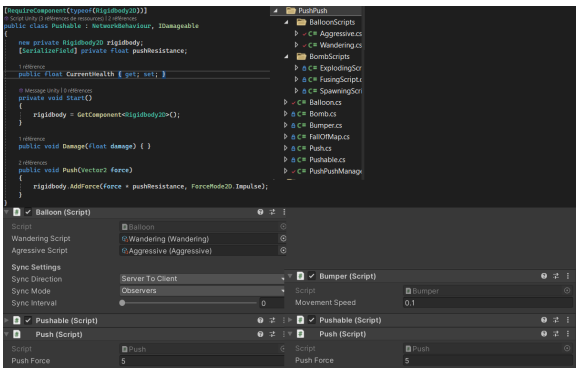


FIGURE 4 – Scripts cours munis d'interface, nombreux et réutilisables

1.3.2 Problèmes initialement rencontrés

Les problèmes rencontrés par rapport aux changements d'architecture et de hiérarchie du projet étaient majoritairement liés au travail fait chacun de son côté il a donc fallu insister sur une hiérarchie plus stricte. Lors du raccord de deux branches les codes ne suivaient pas la même logique il a donc fallu changer quelques composants sur les prefabs. Dans tout les cas beaucoup de refactorisations ont été nécessaires.

1.3.3 La nouvelle architecture après la première soutenance

La nouvelle architecture du jeu nous à apporté une manière plus simple et moins relié au **PlayerBase** de créer des jeu.

1.3.4 GameManagers

Les game managers sont au centre de la création de nouveaux jeux, plus besoin de passé par plusieur switch case douteux comme celui ci :

```
private void ReloadPlayerStarterScript_clientrpc(string scenename)
{
    switch (scenename)
    {
        case "Scene_MainMenu":
            GetComponent<NBPlateformerMovementPlayer>().enabled = false;
            GetComponent<SpriteRenderer>().enabled = false;
            GetComponent<BarrelTurning>().enabled = false;
            GetComponent<BoxCollider2D>().enabled = false;
            GetComponent<BarrelTurning>().barrel.SetActive(false);
            // GetComponent<BarrelTurning>().Tank_Bottom_Part.SetActive(false);
            // tempo
            GetComponent<InputManager>().enabled = false;
            break;
        case "Scene_Level1":
            GetComponent<NBPlateformerMovementPlayer>().enabled = true;
            GetComponent<SpriteRenderer>().enabled = true;
            GetComponent<SpriteRenderer>().material.SetColor("_PartColor", TeamColorSystem.HTMLToColor(Team));
            GetComponent<BarrelTurning>().enabled = false;
            GetComponent<BoxCollider2D>().enabled = true;
            GetComponent<BarrelTurning>().barrel.SetActive(false);
            // GetComponent<BarrelTurning>().Tank_Bottom_Part.SetActive(false);
            // tempo
            GetComponent<InputManager>().enabled = false;
            break;
        case "Scene_TankMinigame":
            GetComponent<NBPlateformerMovementPlayer>().enabled = true;
            GetComponent<SpriteRenderer>().enabled = false;
            GetComponent<BoxCollider2D>().enabled = true;
            GetComponent<BarrelTurning>().enabled = true;
            GetComponent<BarrelTurning>().barrel.SetActive(true);
            // GetComponent<BarrelTurning>().Tank_Bottom_Part.SetActive(true);
            // tempo
            GetComponent<InputManager>().enabled = false;
            break;
        case "Scene_PushPush":
```

FIGURE 5 – Switch case

Nous avons à présent des objet contenant toutes les informations du jeu et une fonction pour finir le niveau :

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public interface IGameManager
{
    public GameObject PlayerPrefab { get; }
    public CustomTimer Timer { get; set; }

    public Vector2 PositionP1 { get; }
    public Vector2 PositionP2 { get; }


    public void EndOfLVl();
}
```

FIGURE 6 – Game manager

Les créateur de jeu peuvent alors ajouter leurs jeux sans problème et renseigner des information comme la prefab du joueur (à présent complétement détaché du joueur) le **CustomTimer** pour get et set le temps et la position initiale des joueurs 1 et 2.
Ils peuvent en plus rajouter leur propre logique par dessus.
Représentation UML en bas de partie.

1.3.5 IPlayer

IPlayer est une interface utilisée en adéquation avec les game managers elle est présente pour définir le player et lui donner des infos que seule la **PlayerIdentity** ou **PlayerPrefab** possède. *Représentation UML en bas de partie.*

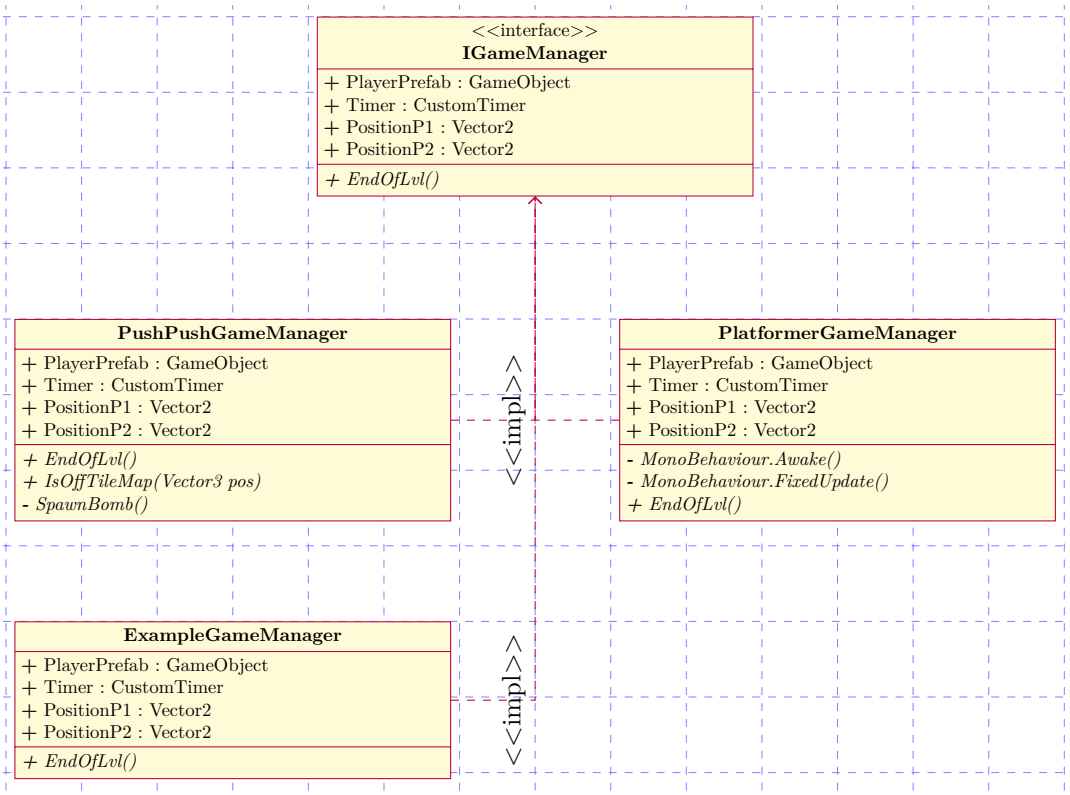


FIGURE 7 – Game managers

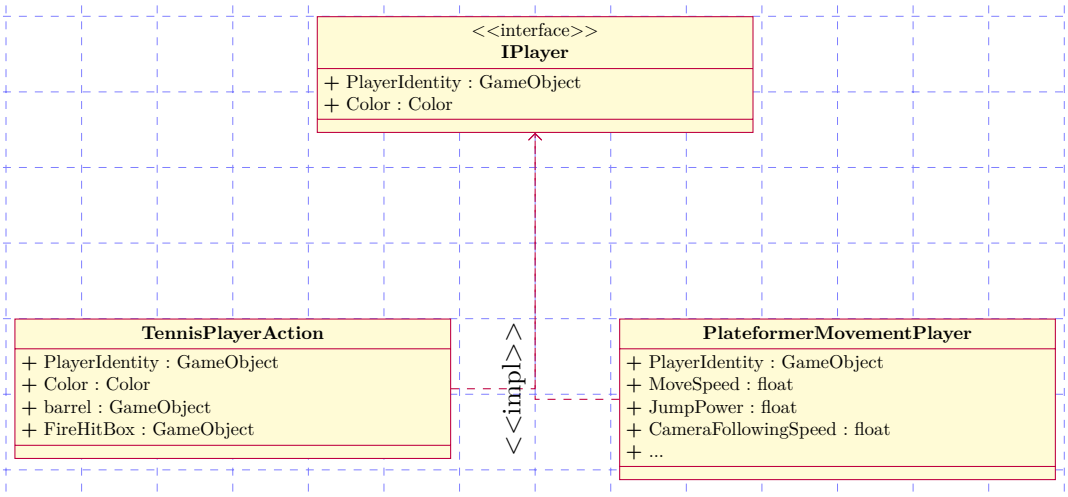


FIGURE 8 – Players

1.4 Les Minis Jeux

Dans cette partie nous parlerons des règles de jeux qui fondront notre gameplay et de différents mini-jeux.

1.4.1 Règles du jeu

Les parties se dérouleront par sections de 5 mini jeux, 4 jeux qui permettront aux deux joueurs de se confronter avec la possibilité de perte de vie et 1 mini jeu mélangeant plusieurs mini jeux considéré comme "boss". Les sections de mini jeux seront de plus en plus difficiles et il sera possible de continuer à jouer de manière infinie et ainsi d'augmenter toujours plus la difficulté, notamment le temps autorisé pour réaliser le mini jeu. Certains paramètres comme le choix des mini jeux, le nombre de vie ou la difficulté pourront être choisis par les joueurs.

1.4.2 Tank Minigame

Inspiré principalement de **Wii Party**, ce Mini Jeu est compris dans beaucoup de Party games pour son aspect compétitif et ludique. Cette bataille confrontera donc 2 tanks controllé par les joueurs. Ceux-ci devront tirée sur le tank adversaire et le détruire avant la fin du temps impartis, sans quoi il y aura égalité, le tout en esquivant les tirs de tourelles non-joueur (dont l'on parlera plus en détail dans la partie Intelligence artificielle).

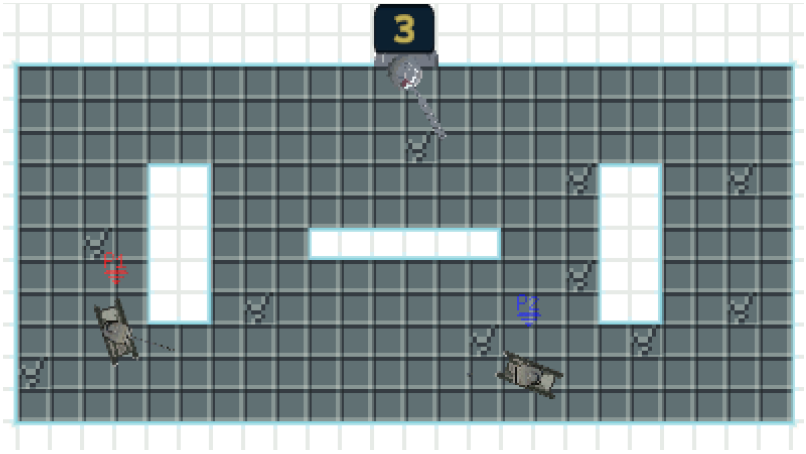


FIGURE 9 – Tank Game

1.4.3 Push Minigame

Ce jeu est inspiré du jeu d'arcade **Motos** de Namco. Il consiste à pousser le joueur adverse hors de la map, hors une horde d'énemies sous forme de ballons voudront également vous éliminer, sans compter les bombes placé aléatoirement sur le terrain pour vous poussé et enlever des tiles.

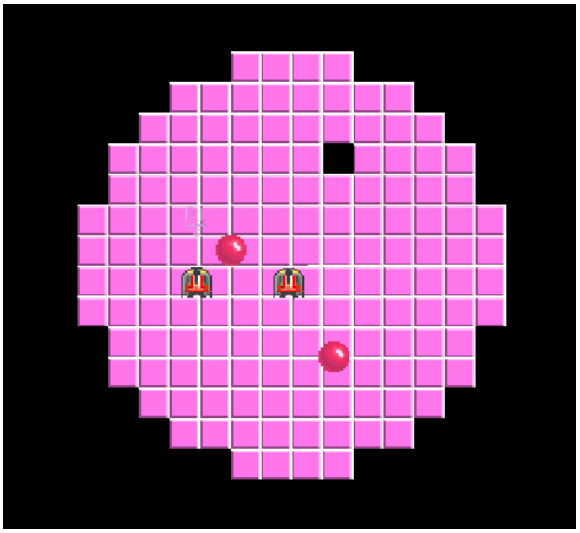


FIGURE 10 – Pusher Game

1.4.4 Plateformer MiniGame

Plateformer Game est, par son apparence, un jeu très simple à la **Super Mario Bros**, on saut en esquivant des ennemis, pour atteindre la fin du niveau. Cependant, on ne peut pas vraiment le qualifier de platformer basique non plus, car en effet, le jeu n'est pas qu'un parcours, c'est une course, à la fois contre la montre, et contre l'autre joueur. Le premier qui arrive à monter jusqu'à l'arrivée gagne.

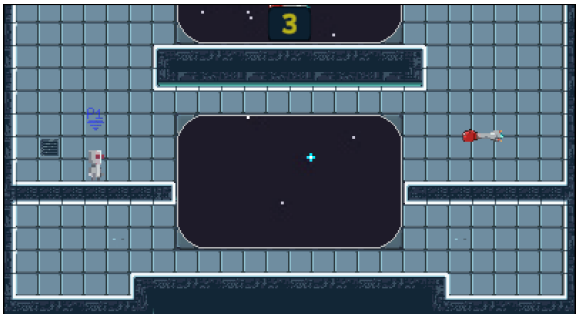


FIGURE 11 – Plateformer Game

Vous pouvez des exemples sur l'evolution des sprites du joueur de ce mini-jeu, dans l'annexe (cf : FIGURE 31, 32 et 33)

Bien sur, ce jeu comprends des ennemis, qui déran fggeront les 2 utilisateurs dans leur course. En effet, des missiles à tête chercheuses s'élanceront sur les joueurs, mais ne font que pousser ceux-ci, car leurs têtes explosives sont recouvertes de gants de box.

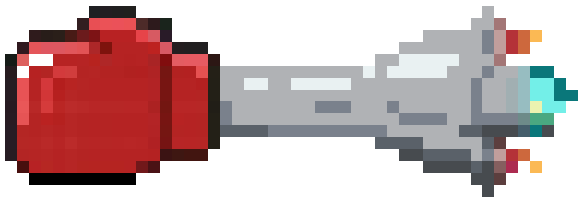


FIGURE 12 – L'ennemi en question

Nous parlerons plus tard précisemment de l'intelligence artificielle derrière cet ennemis.

1.4.5 Spatialship game

Spatialship game est l'un de nos jeu les plus ambitieux, il proposera une phase de profil dans laquelle il faudra tirer des rayons laser avec notre vaisseau sur le boss qui sera une IA. De plus nous pourrons tirer sur le joueur adverse pour bloquer ses mouvements. Par la suite après une phase de profil, le jeu basculera en vue de face et il faudra esquiver les obstacles lancés par le boss tout en tirant sur ses points faibles en cliquant dessus. Il sera également possible de rentrer volontairement dans des obstacles pour obtenir un bonus qui permettra de nuire aux actions du joueur adverse.

1.4.6 Volley Game

Ce Mini-jeu reprend le principe du Volley-ball, il faut réussir à faire tomber la balle dans le terrain de son adversaire, dans un temps imparti. Cependant, les joueurs ne controle pas un sportif, il controle une machine capable d'être des ondes de choc, pour renvoyer le balon.

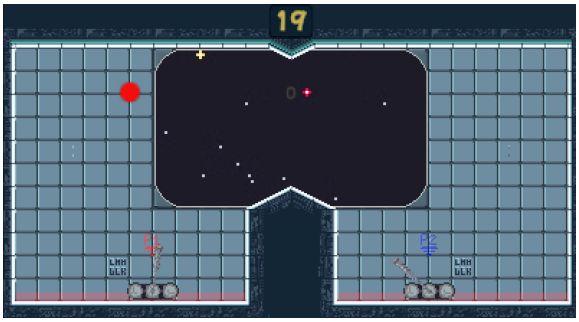


FIGURE 13 – Volley Game

1.5 L'interface utilisateur

1.5.1 Menu de connexion multijoueur

Sur le menu multijoueur, le premier élément est un bouton pour lancer en mode host. Le second est un bouton pour rejoindre en tant que client un serveur, le troisième est une case pour définir l'adresse IP du serveur que l'on veut rejoindre, et le dernier élément est une case pour définir la team du joueur. Lors de la connexion en tant qu'hôte ou en tant que client, les boutons sont bloqués pour empêcher tout type de manipulation durant la connexion qui pourrait créer un bug.

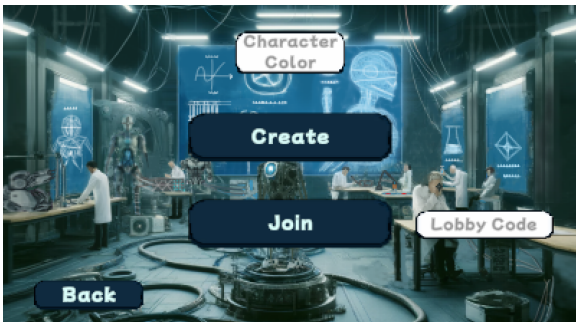


FIGURE 14 – Menu de connexion

1.5.2 Sélection de mini-jeux

La sélection personnalisée de mini-jeux permet aux joueurs de sélectionner un ou des niveaux spécifiques, et ainsi de pouvoir jouer dans les mini-jeux qu'ils préfèrent. Elle sera disponible au même endroit que le lancement de mini jeux aléatoires, après avoir lancer le salon de partie.

1.5.3 La couleur de la Team

Les différentes couleurs associées aux différents joueurs sont décidées par les joueurs lors de leur première connexion au serveur. Dans la case à remplir pour définir la team, nous avons décidé de transformer tout type de texte en code couleur HTML pour permettre de définir la couleur que l'on veut selon un code spécifique.
Ce système de conversion se déroule de la manière suivante. Tout charactère minuscule est traduit en majuscule, tout charactère majuscule de A à F ou nombre est laissé comme telle, tout autre éléments est traduit avec un modulo en un nombre si il est inférieur à la valeur de "A" et est traduit avec un modulo en une majuscule de A à F pour un nombre supérieur à la valeur de "A". À la fin, il est rajouté un "" au début du string.Il est possible dans la case à remplir de prévisualiser cette couleur pour plus de compréhension.

FIGURE 15 – Case de changement d'équipe

1.6 Le multijoueur

1.6.1 Netcode for GameObjects

Lors de la conception de ce projet, dès la remise du premier cahier des charges, nous nous sommes directement intéressés à l'aspect Multijoueur. En effet, ce projet avait pour condition d'inclure du multijoueur, et s'il n'était pas implémenté en premier, il faudrait ensuite refaire tous le code pour l'y intégrer.

En vue du défi presque impossible qu'est la conception de son propre système multijoueur, nous avons opté pour l'utilisation de "Netcode for GameObjects", un système de multijoueur intégré de base à Unity.

Cependant, ce système ne correspondait pas à notre projet, car en effet, Netcode for GameObjects est un système multijoueur utilisé principalement pour faire des jeux où les utilisateurs doivent coopérer, ne nécessitant généralement pas de gérer les problèmes d'inégalités de connexion entre les joueurs. Il nous fallait donc un système plus optimisé pour le face à face entre joueurs.

1.6.2 Mirror

Il fallait donc recommencer sans système host/client (de Netcode), et nous nous sommes penchés plutôt sur Mirror qui est hérité de UNET et qui était une ancienne technologie de Unity, qui permet d'avoir un système host/client avec un host qui contient le serveur mais également qui est un client, ce qui simplifie grandement le code.

Par ailleurs, la prise en main de Mirror offre également quelque avantages comme par exemple la possibilité de demander directement d'exécuter une fonction sur serveur ou sur tous les clients avec [Command] et [ClientRpc]. Il permet également tout comme Netcode de synchroniser les variables entre le serveur et le client avec [SyncVar] que ce soit du serveur vers les clients ou du client vers le serveur.

1.6.3 Unity Relay et Unity Transport Package

Cependant, un dernier problème subsistait concernant le multijoueur de Another Recursion, en effet, la difficulté de connexion à son adversaire / partenaire. Avant cette modification, chaque joueur nécessitait d'avoir un accès à son pare-feu, et devait donner toutes autorisations réseau à l'application, puis le créateur de la partie devait trouver son IP, pour que l'autre la mentionne dans le jeu.

Quoique que cela ne soit pas un problème en soit, car la plupart des jeux en LAN passent par ce processus, cela pose un problème concernant notre cible de joueurs. En effet, les enfants (par exemple) ne savent peut-être même pas ce qu'est un pare-feu, et sont probablement restreints par un code parental.

Nous avons donc choisi une interface de connexion plus simple pour le joueur, Unity Relay (en utilisant Unity Transport Package). Il permet de créer une partie sur les serveur d'Unity en utilisant pour s'y connecter qu'un code de 6 caractères. Cela nous a permis de garder notre système mutlijoueur local (mirror) tout en facilitant la facilité de connection entre les joueurs.

1.6.4 Installation du serveur et des clients

L'installation du système multijoueur n'a pas été une tâche facile mais il est complètement implémenté et il ne risque pas de changer grandement. Tout d'abord, nous n'avons pas eu de mal à avoir un bouton qui crée un hôte autrement dit, un serveur et un client connectés au serveur, à avoir un bouton qui rejoint le serveur connecté à l'adresse de la partie écrite dans la case.

Cependant pour résoudre tout problème qui pourrait survenir et qui sont arrivés à de nombreuses reprises durant le développement du projet, nous avons créé de nombreux systèmes. Premièrement, un système qui bloque toutes interaction avec l'interface, durant la connexion de notre joueur, pour empêcher tout bug de survenir. De plus, nous avons également créé un système qui tente à de nombreuses reprises de se connecter

et nous pourrons peut-être à l'avenir mettre dans les paramètres généraux le choix des délais de connexions. Par ailleurs, nous avons aussi bloqué toutes connexions de client au serveur lorsqu'il est dans le chargement de création du serveur mais aussi lorsque le jeu est lancé et que le salon autrement appelé lobby est quitté.

1.6.5 Installation du système de Team sur le serveur

Pour différencier les équipes de joueurs même si le jeu se concentrera sur le 1 contre 1, nous avons décider de proposer au joueur dès l'interface de connexion la possibilité de choisir une team. Cette team est un code couleur HTML pour simplifier son utilisation pour les graphismes. Durant le chargement de connexion, la team envoyée par le client est associée au GameObject du joueur sur le serveur puis envoyé à tous les clients avec [SyncVar].

1.6.6 Chargement des joueurs entre les scènes

Le chargement des joueurs entre les scènes a été une tâche très complexe et semée d'embûches. En effet, par défaut le joueur est supprimé à chaque scène, c'est pourquoi nous avons décidé de placer dans sa fonction de début la fonction "DontDestroyOnLoad(gameObject) ;" pour empêcher le joueur de maintenir son état pour les scènes suivantes.
Comme nous gardons le joueur, nous devons donc initialiser le GameObject du joueur à chaque scènes correspondant à chaque mini-jeux. Ainsi, à l'aide d'un bouton start, il est exécuté une fonction sur le joueur hôte de la partie qui va passer à la prochaine scène dans la liste des scènes qui sont triées dans un ordre précis. Dans un avenir proche, nous implémenterons un système qui prendra les sections de 5 mini-jeux et de façon aléatoire en lancera une puis une autre jusqu'à la mort d'un joueur ou jusqu'à arrêt volontaire. De plus, avant de charger la scène, les paramètres pour la scène sont chargés sur le GameObject de tous les joueurs sur tous les clients

1.6.7 Synchronisation des Input

L'une des dernières améliorations que nous avons apporté à notre jeu est la synchronisation des input du clavier ou de la manette. Le système fonctionne de la manière suivante, toutes les millisecondes, les touches du clients modifient la variable correspondant à leur touche en attribut et ces valeurs sont synchronisées avec le serveur grâce à [SyncVar] qui est mit en mode envoi de données du client vers le serveur. Ainsi le serveur peut parfaitement lire chacune des touches des différents joueurs et ainsi exécuter une action spécifique.

1.7 L'intelligence artificiel

Comme cité précédemment, certains de nos mini jeux contiennent certainnes formes d'intelligence artificielle. Cette partie sera dédié à l'étude de celle-ci.

1.7.1 Debut de l'IA des PNJ ennemis du Tank Game et du Platformer GAme

Les IA des ennemis du Tank Game et du Platformer Game commencent par un même système, la recherche de chaque joueur présent dans la partie. Cela ne se passe bien sur que sur le serveur.

```
private void UpdateTarget()
{
    if (allTargets.Length == 0)
        allTargets = GameObject.FindGameObjectsWithTag("UnderPlayer");
    GameObject nearestObj = null;
    float shortestDistance = Mathf.Infinity;
    foreach (GameObject g in allTargets)
    {
        float f = Vector3.Distance(a: g.transform.position, b: transform.position);
        if (shortestDistance > f && CanSeeTarget(g))
        {
            shortestDistance = f;
            nearestObj = g;
        }
    }

    if (nearestObj is not null && shortestDistance <= range)
        target = nearestObj.transform;
    else
        target = null;
}
```

```
private bool CanSeeTarget(GameObject g)
{
    RaycastHit2D raycast = Physics2D.Raycast(origin: transform.position,
        direction: g.transform.position - transform.position,
        distance: Mathf.Infinity, layerMask: 1);
    if (raycast.collider.gameObject == g)
        return true;
    return false;
}
```

Ce programme permet de trouver tous les joueurs, puis de vérifier si il sont dans la distance de détection de l'ennemi, puis un vecteur entre l'ennemi et le joueur est tracé, et si cette ligne ne traverse pas de mur, alors le joueur est définit comme étant la cible de cette IA.

1.7.2 Reste de l'IA des PNJ ennemis du Tank Game

L'IA de la tourelle après avoir trouvé une cible avec le programme montré ci-dessus, va ensuite envoyée un projectile vers cette cible. Une fois cela fait, un décompte va commencer empèchera la tourrelle de tirer, pour simuler un temps de rechargement.

1.7.3 Reste de l'IA des PNJ ennemis du Platformer Game

L'IA de l'ennemi du missile boxer est un peu plus complexe que celle vu juste au dessus. Elle reprend le même début pour trouver une cible, mais ensuite commence un tant de préparation, au niveau animation, qui montre que l'IA a trouvé une cible, puis va foncer sur la position trouvée. Cependant, même si le joueur bouge, l'ennemi continuera d'avancer dans cette direction jusqu'à rentrer dans un mur, ou un autre joueur.

1.7.4 Intelligence artificielle dans PushPush

L'IA contenu dans le mini-jeu PushPush contient une **"Finite State Machine"**, un principe en orienté objet qui se base sur une class contenant des **"States"** par compositions qui peuvent ensuite être échanger, coupler avec le système de **ScriptableObject** (objet chargé dans les assets contenant un unique script) de unity cela permet de faire des State-Machines modulables ou les scripts peuvent être remplacé par n'importe quelle autre scripts et permettre des changement de comportement distinct.

L'IA des balloons PushPush est basé sur deux états **Aggressive** et **Wandering**.
Lors de sa phase Wandering le balloon ne fait que érrer dans des dirrections aléatoires.

```
Script Unity | 0 références
public class Wandering : State
{
    private Transform transform;
    private float timer;
    private IAI ai;

    3 références
    public override void Enter(StateMachine stateMachine)
    {
        if (stateMachine.Behaviour is IAI initAi)
            ai = initAi;
        transform = stateMachine.Behaviour.transform;

    }
    public override void Update()
    {
        // Debug.Log("I am wandering");

        timer += Time.deltaTime;

        if (transform is null)
            return;

        if (timer >= 5f)
        {
            SetNewTargetPosition();
            timer = 0;
        }

        Vector3 direction = (ai.RandomTarget - transform.position).normalized;


        transform.position += 2f * Time.deltaTime * direction;
    }

    1 référence
    private void SetNewTargetPosition()
    {
        Vector3 randomDirection = Random.insideUnitSphere * 5f;
        randomDirection += transform.position;
        randomDirection.y = transform.position.y; // Keep the y-axis level

        ai.RandomTarget = randomDirection;
    }
}
```

FIGURE 16 – Wandering State

Lors de la phase Aggressive le balloon s'approche vers le joueur pour essayer de la pousser.

```
[CreateAssetMenu(menuName = "MyScriptables/Aggressive")]
Script Unity | 0 références
public class Aggressive : State
{
    private Transform target;
    private Transform self;

    3 références
    public override void Enter(StateMachine stateMachine)
    {
        if (stateMachine.Behaviour is IEnemy enemy)
            target = enemy.Targeted;
        self = stateMachine.Behaviour.transform;
    }
    2 références
    public override void Update()
    {
        self.position = Vector3.MoveTowards(self.position, target.position, Time.deltaTime * 2f);
    }
}
```

FIGURE 17 – Aggressive State

L'entité change d'état lorsque le joueur sort et rentre d'un **Circle-Collider** posé sur le balloon.

```
private void OnTriggerEnter2D(Collider2D collider)
{
    if (!isServer)
        return;

    if (collider.gameObject.GetComponent<IPlayer>() is null)
        return;

    targeted = collider.transform;
    stateMachine.ChangeState(AgressiveScript);
}

Message Unity | 0 références
private void OnTriggerExit2D(Collider2D collider)
{
    if (!isServer)
        return;

    stateMachine.ChangeState(WanderingScript);
}
```

FIGURE 18 – Changes State

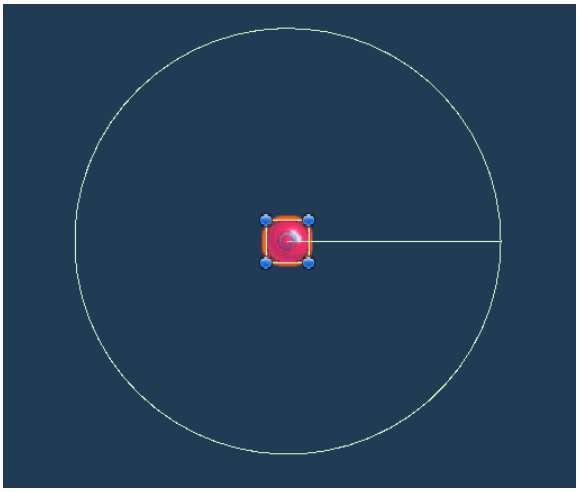


FIGURE 19 – Aggressive zone

Voici une représentation UML de cette StateMachine :

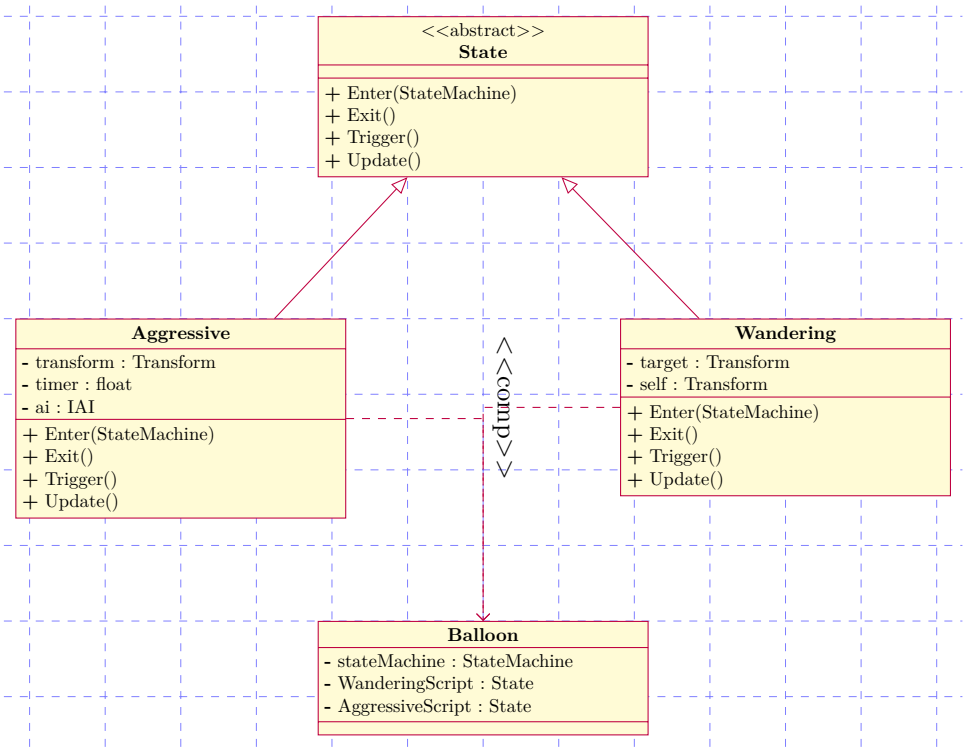


FIGURE 20 – Ai State Machine

De plus pour récupérer des informations complémentaires nous pouvons passer par de nombreuses interfaces :

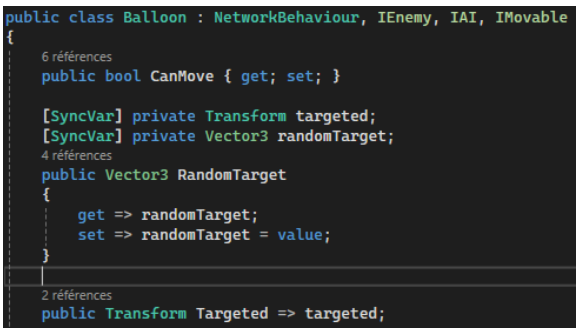
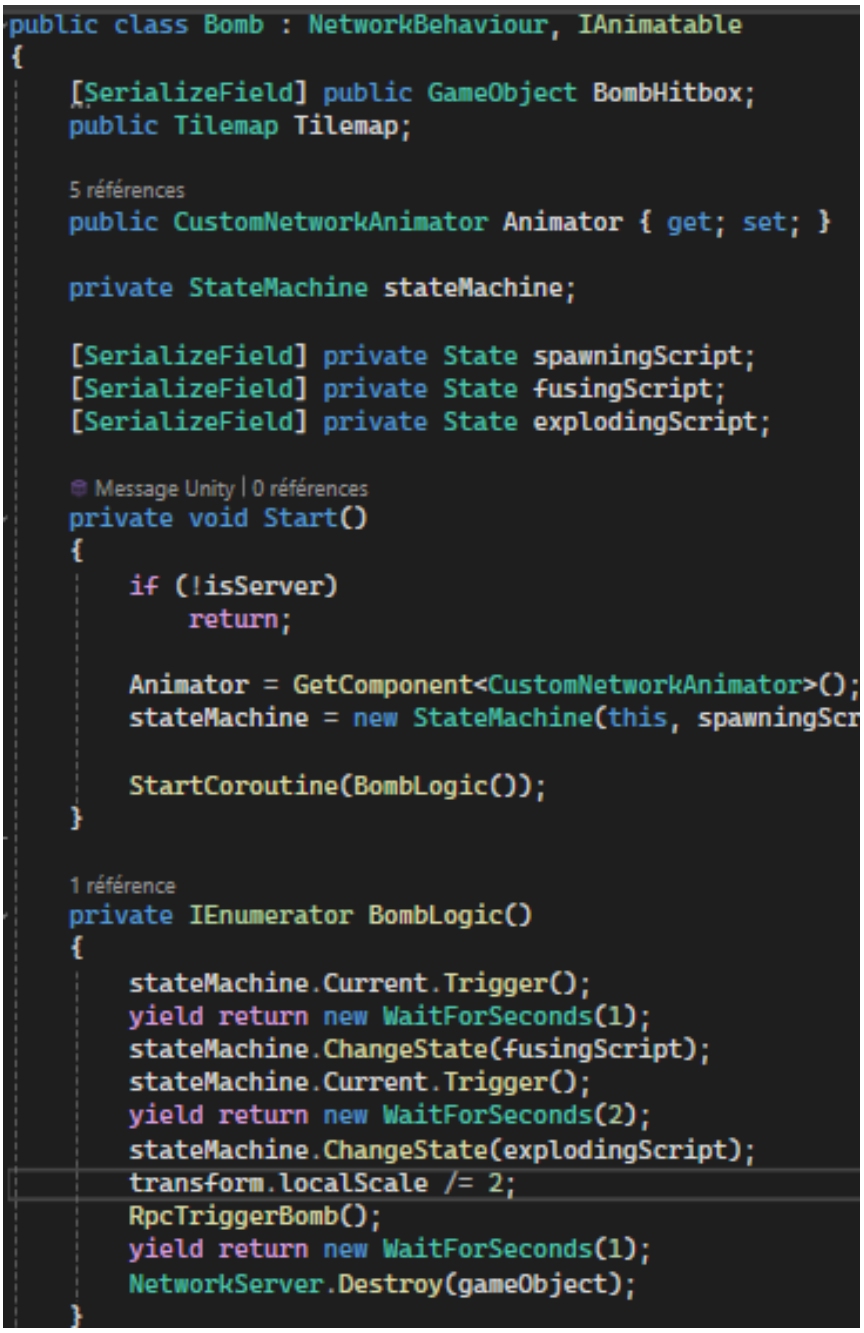


FIGURE 21 – Interfaces

Cette approche à aussi été utiliser pour le code des bombes de manière à ne pas passer par la gestion cahotique des animations. La dernière ne contient pas que une animation mais aussi la logique d'éxplosion devant être syncronyser sur les clients



```
public class Bomb : NetworkBehaviour, IAnimatable
{
    [SerializeField] public GameObject BombHitbox;
    public Tilemap Tilemap;

    5 références
    public CustomNetworkAnimator Animator { get; set; }

    private StateMachine stateMachine;

    [SerializeField] private State spawningScript;
    [SerializeField] private State fusingScript;
    [SerializeField] private State explodingScript;

    Message Unity | 0 références
    private void Start()
    {
        if (!isServer)
            return;

        Animator = GetComponent<CustomNetworkAnimator>();
        stateMachine = new StateMachine(this, spawningScr

        StartCoroutine(BombLogic());
    }

    1 référence
    private IEnumerator BombLogic()
    {
        stateMachine.Current.Trigger();
        yield return new WaitForSeconds(1);
        stateMachine.ChangeState(fusingScript);
        stateMachine.Current.Trigger();
        yield return new WaitForSeconds(2);
        stateMachine.ChangeState(explodingScript);
        transform.localScale /= 2;
        RpcTriggerBomb();
        yield return new WaitForSeconds(1);
        NetworkServer.Destroy(gameObject);
    }
```

FIGURE 22 – Bomb StateMachine

1.7.5 Intelligence artificielle de choix de niveau

De plus, nous pensions inclure une intelligence artificielle gérant le choix de mini-jeux et leur difficulté. En effet, pour que notre jeu ne soit pas trop répétitif, il faudrait que le même jeu ne soit pas utilisé 2 fois de suite, qu'il y ait certaines variations entre chaque challenge, et aussi, si le mini-jeu possède une difficulté variable, de la choisir en fonction du temps de jeu de la partie des joueurs, permettant une augmentation de la difficulté avec le temps.

1.8 Les graphismes

1.8.1 Création de base des personnages

Étant donné que nous sommes relativement nouveaux dans le domaine du pixel art, nous avons consacré du temps à explorer et à nous familiariser avec les techniques, les outils et les ressources disponibles.

Nous avons étudié divers tutoriels en ligne, examiné des œuvres de jeux en pixel art tel que Pokémon et Stardew Valley , et nous avons expérimenté différents logiciels spécialisés dans le pixel art. Nous avons donc opté pour l'utilisation d'Aseprite, un logiciel permettant de créer différentes frames mais nous avons aussi utilisé paint.net pour la création de certains sprites, comme les lettres composant le nom ANOTHER RECURSION. Nos sprites sont donc en Pixel Art, ce qui permet de les importer sous forme de Gifs.

Notre but premier a été tout d'abord de créer la forme de base des personnages qui constituerons notre jeu. Nous avons alors opté sur un format 16-bits pour apporter assez de détails mais aussi pour limiter la complexité des animations. Nous nous sommes inspirés des jeux gameboy pour produire l'animation, tandis que pour les couleurs et le design des personnages, nous allons faire en sorte qu'ils conviennent au thème futuriste et robotique de notre jeu.

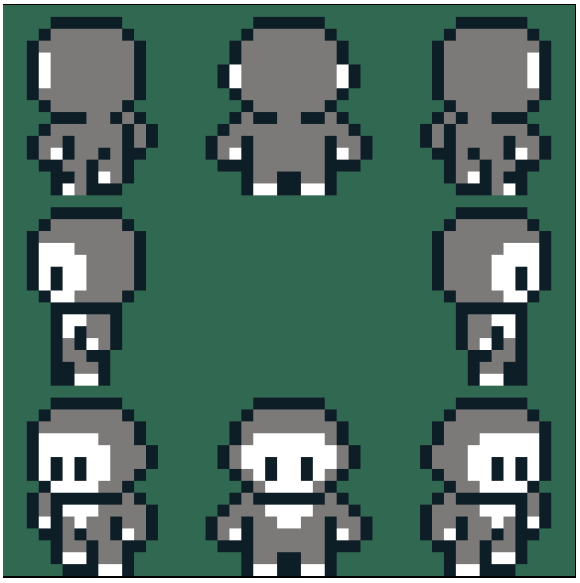


FIGURE 23 – Premier sprite : Sprites du petit robot

1.8.2 Evolution des graphismes

Une fois le commencement de la programmation des jeux, la nécessité de "Place Holder", des images qui ne restent pas dans la version finale du jeu, ont été faite.
Exemple : Ce sprite a été utilisé comme sprite de logo en premier temps, avant d'être remplacer par le logo actuel :

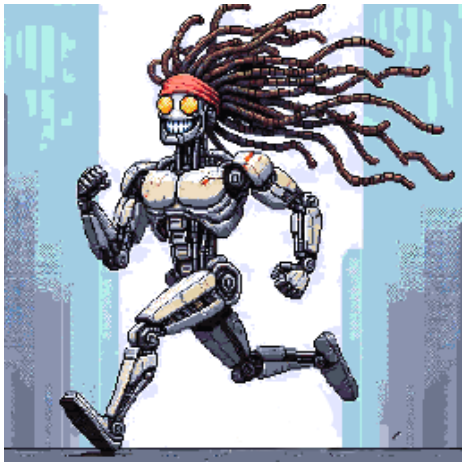


FIGURE 24 – Première version de l'icône du jeu, faite par IA

Par inspiration, nous avons aussi fait des images "Place Holder" pour les animations, tel que cette robot qui cours.

FIGURE 25 – Sprite original de la course du robot

Cependant, ce robot ne respectait exactement notre vision tout publique (et un peu enfantine) de notre jeu, sans cité le fait qu'une animation n'est pas constitué d'un seule image, mais de plusieurs. C'est à ce moment qu'ont été produits les premier "schémas d'animation" permettant de réutiliser ces images pour plusieurs graphismes différents :

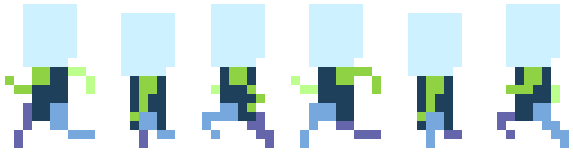


FIGURE 26 – Une animation schéma de l'animation de course

Et une fois peaufiné, le résultat donne quelque chose de convainquant :

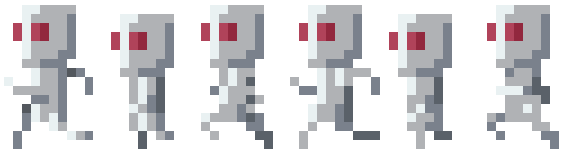


FIGURE 27 – L'animation de course finale du joueur dans le mini jeu "Plateforme"

Cette méthode de pensée, et de production d'image nous a permis d'avancer en même temps sur les mini-jeux, et sur les graphismes, permettant au développeur de faire les animations initiales et de travailler avec pour résoudre d'éventuels défauts, puis ensuite au graphistes de refaire ces animations avec les vrai images, et de les implémenter.

Cela permet additionellement de faire travailler les responsables des graphismes sur du code, et non pas uniquement l'implémentation d'image unique.

1.8.3 Conception des boutons

L'interface utilisateur constitue un élément essentiel de toute expérience de jeu, et la conception de boutons clairs et intuitifs est un aspect crucial de cette interface. Pour répondre à cette exigence, nous créerons dans un futur proche une série de boutons pixel art pour les différentes options du menu, telles que "Play" , "Options" et "Exit".

Chaque bouton sera soigneusement conçu pour être à la fois esthétique et fonctionnel. Nous opterons pour des formes simples et des couleurs adaptées à notre thème futuriste afin de garantir leur lisibilité pour les joueurs. En accord avec le style pixel art du jeu, nous veillerons à ce que l'animation des boutons soit esthétiquement plaisante mais aussi adaptée à notre jeu.

FIGURE 28 – Premier sprite du bouton play

1.8.4 Création du décor

Le décor joue un rôle essentiel dans la création de l'ambiance et de l'immersion d'un jeu vidéo. Dans notre projet, nous consacrerons du temps à la conception d'arrière plans pixel art pour qu'ils soient futuristes et agréables.

Nous nous sommes inspiré d'images libre de droit d'auteurs pour produire les premiers décors. Chaque élément du décor a été soigneusement pensé pour contribuer à l'histoire et à l'univers du jeu. Nous avons travaillé sur l'agencement des éléments, les variations de couleurs et certains petits éléments présents sur l'arrière-plan, ce qui permet de renforcer l'ambiance futuriste d'Another Recursion. Chaque élément du décor est conçu pour raconter une histoire et pour stimuler l'imagination des joueurs, contribuant ainsi à créer une expérience de jeu captivante et mémorable.

On peut notamment citer l'utilisation de TileSet, un assemblage d'image dont nous parlerons dans la prochaine partie.

1.8.5 Résolution, Pixel Perfect Camera et TileSet

Au niveau de la résolution de notre jeu nous avons choisi d'utiliser la pixel perfect camera afin de restreindre tout pixel à l'écran à une résolution fixe et de permettre corriger visuellement automatiquement tout élément qui ne serait pas en pixel art. Pour la résolution fixe nous avons choisi de nous baser sur le format le plus commun qui 1920 par 1080 autrement dit un ratio de 16 par 9.

Nous avons décidé de le diminuer pour avoir un jeu plus pixelisé, ainsi la résolution choisie fut dans un premier temps 400 par 225. Par la suite étant donné nos tile map qui sont fixées sur une grille de 16 pixels par 16 pixels nous avons choisi la résolution 400 par 224 pour permettre d'avoir deux multiples de 16 et ainsi permettre aux tile map de mieux se coller à notre résolution.

Nous avons donc pu en produire des tileset, des assemblage d'image qui permettent de créer un environnement à partir de ces pieces d'étachés de décors, murs, et bien d'autre.

Par exemple : ceci est un tile set basique de mur :

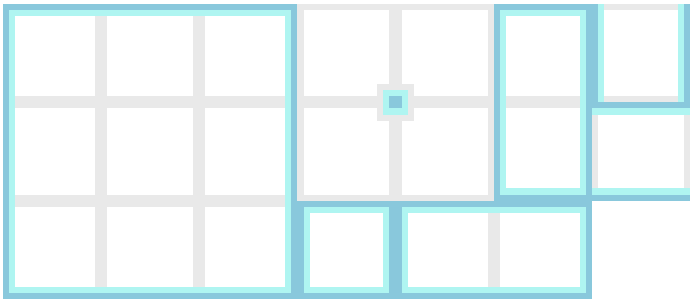


FIGURE 29 – Tile palette des murs / sol

Il permet, avec quelque autre TileSet citée dans l'annexe, de faire ce genre d'environnement :

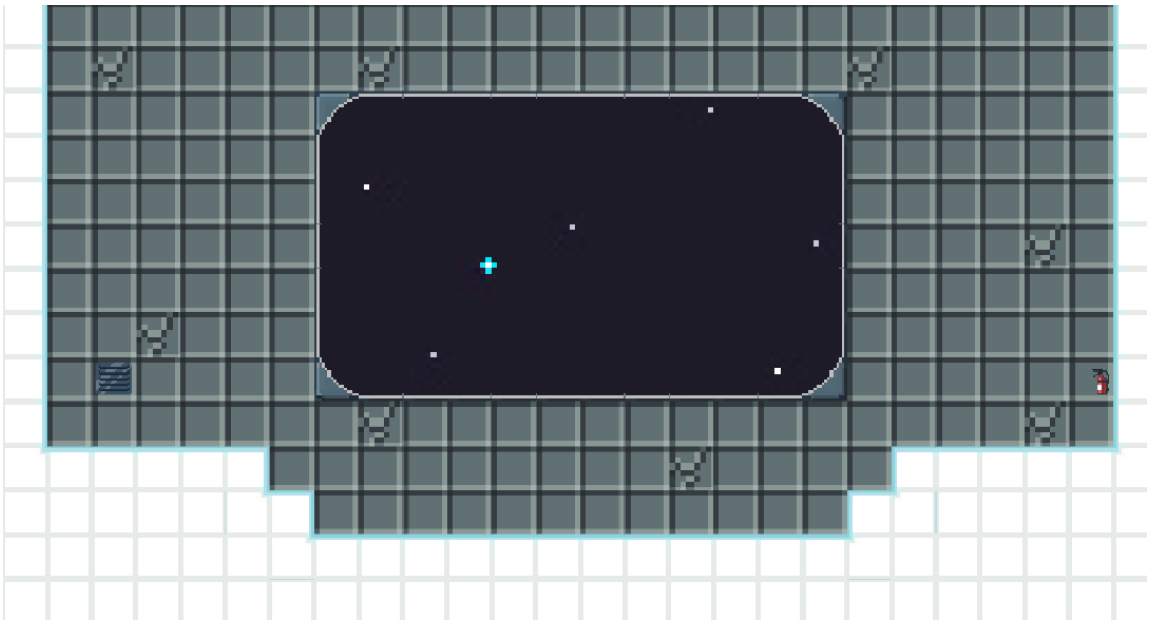


FIGURE 30 – Exemple d'ambience du jeu

1.8.6 Shader Graph

Lors de la conception des équipe, pour différencier chacune des équipes nous souhaitions rajouter une possibilité de modifier certaines des parties du sprite du joueur, ainsi pour le sprite du robot nous avions voulu modifier la couleur du bandeau en fonction de la couleur de l'équipe. Pour ce faire nous avons décidé d'utiliser un shader personnalisé.

Pour nous simplifier la tâche, nous avons utilisé l'asset Shader Graph qui permet de créer un shader avec des outils visuel. Comme nous ne sommes qu'au début de la création de sprite, pour l'instant le shader se limitera qu'à juste rajouter au dessus de notre sprite actuel une version du bandeau du robot avec une variable pour sa couleur qui sera définie à chaque fois que le sprite du robot apparaîtra.

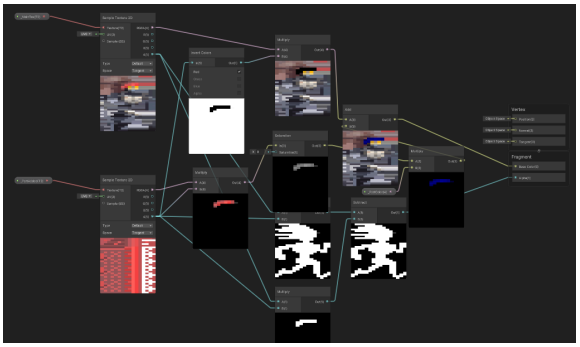


FIGURE 31 – Shader Graph du shader pour la couleur de l'équipe

1.9 Le son

1.9.1 La musique

La musique joue un rôle crucial dans la création de l'ambiance et de l'immersion d'un jeu vidéo. Notre équipe a entrepris de composer une bande-son originale qui captivera les joueurs et les plongera dans l'univers futuriste et robotique du jeu. Nous avons commencé par définir l'ambiance et le ton général du jeu, en tenant compte de son histoire, de son esthétique et de son gameplay. Nous avons utilisé une application de mixage et de créations appelée FL Studio pour permettre de créer des mélodies adaptées.

En nous appuyant sur ces éléments, nous avons composé des morceaux musicaux qui reflètent ces aspects. Nous avons exploré différents genres musicaux, des mélodies épiques aux ambiances atmosphériques, en cherchant à créer une bande-son variée et dynamique qui accompagnera efficacement les différents moments du jeu. Notre objectif était de créer une musique qui soit à la fois mémorable et évocatrice, et qui contribue à renforcer l'immersion des joueurs dans l'univers du jeu, en s'inspirant parfois de bandes sons connues telles que celles de Minecraft.

On a alors utilisé un système de boucle musicale, pour créer une musique qui est certes répétitive mais adaptée aux différents niveaux de notre jeu.

1.9.2 Les effets sonores

Les effets sonores sont un autre élément essentiel pour créer une expérience de jeu immersive. Nous avons créer une large gamme d'effets sonores, des bruits ambiants aux sons d'action (par rapport à notre jeu), en passant par les bruits de pas et d'attaques des personnages.

Nous savons créé des effets sonores de haute qualité qui correspondent à l'esthétique et à l'atmosphère du jeu. Nous avons exploré différentes techniques d'enregistrement et de manipulation sonore pour obtenir des résultats authentiques et convaincants. Notre objectif était de créer une palette sonore riche et immersive qui ajoute une nouvelle dimension à l'expérience de jeu.

1.10 Le site web

Dans le cadre de notre projet, nous avons eu pour responsabilité de concevoir et de développer le site web associé au jeu. En utilisant les langages HTML, CSS et JS, nous avons fait de notre mieux pour créer une interface utilisateur fonctionnelle et attrayante offrant aux joueurs un accès facile aux informations essentielles sur le jeu. Ce site rend possible le téléchargement du jeu et des diverses ressources utilisées pour sa création, ainsi que des informations sur le projet, notre équipe et ses mésaventures. Ce site web se devait donc d'être complet.

L'un des aspects clés de la création du site web a été la conception et l'implémentation d'un menu latéral interactif. Ce menu permet aux joueurs de naviguer facilement entre les différentes sections du site, y compris les informations sur le projet, sur l'équipe, et aussi une section depuis laquelle le téléchargement du jeu, du manuel et de ce rapport seraient disponibles. La planification de l'architecture du menu et surtout son implémentation a pris du temps, mais nous nous sommes assurés qu'il était intuitif et facile à utiliser par tous les types de joueurs en plus d'être visuellement agréable.

FIGURE 32 – Exemple de la page d'accueil du site

Nous avons utilisé le CSS, qui est un language permettant la mise en forme des différents éléments du site (paragraphes, images), rendant ainsi le site plus attrayant. Par exemple, lorsque les joueurs survolent les différents éléments du menu et du site, nous avons ajouté des effets d'assombrissement au survol, simple mais efficace, donnant directement l'impression que le site est interactif.

Le JavaScript a également été utilisé pour créer des transitions fluides notamment celles du menu latéral et des effets visuels attrayants.
Les animations plus complexes ont été gérées avec JavaScript.
Nous avons par exemple utilisé des boutons et des EventListener permettant de détecter quand un élément est cliqué, entraînant des changements sur le CSS (le style d'un élément) et donc sur le site web en général. Par exemple le menu latéral est rétractable, il suffit pour cela de cliquer sur l'icône de menu en haut du menu de navigation. Cette implémentation qui a l'air basique peut pourtant prendre des heures à ceux qui ne connaissent pas encore JS.

Grâce à notre travail sur le site web, nous avons pu offrir aux joueurs une expérience immersive qui complète parfaitement notre jeu vidéo, renforçant ainsi la cohérence de notre image.

2 Conclusion

En cette dernière soutenance de projet sur la création de Another Recursion, nous avons, avec un sentiment d'accomplissement, correctement établi les fonctions voulus sur lesquelles notre jeu prend tous ses apsects amusants et intéressants. La mise en place des tous ces mini-jeux, ainsi que l'aspect compétitif, visuel et audio montre une importante progression technique mais également une amélioration de la cohésion de l'équipe malgré notre projet ambitieux de création d'un Party Game avec plusieurs mini-jeux.

Notre équipe, bien que composée de spécialistes dans différents domaines, reconnaît l'importance cruciale de la collaboration et de la polyvalence. Chacun de nous a été appelé à contribuer à la programmation des mini-jeux, illustrant notre engagement envers un développement collaboratif du projet. En cette période de dernière soutenance nous avons pris conscience du chemin parcouru mais aussi des étapes qu'il nous reste à franchir dans le monde de l'informatique et de l'art.

3 Annexe

FIGURE 33 – Sprite du canon du tank

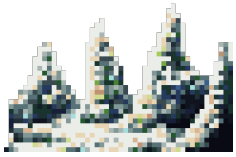


FIGURE 34 – Sprite stalagmite

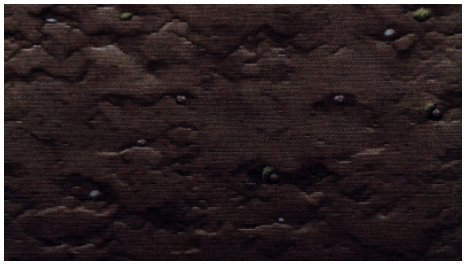


FIGURE 35 – Cave background

FIGURE 36 – Sprite du bandeau robot

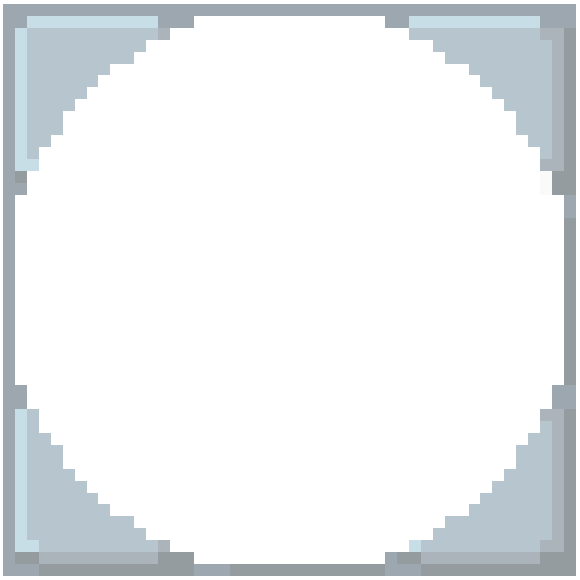


FIGURE 37 – Tile set permettant de faire des fenêtres

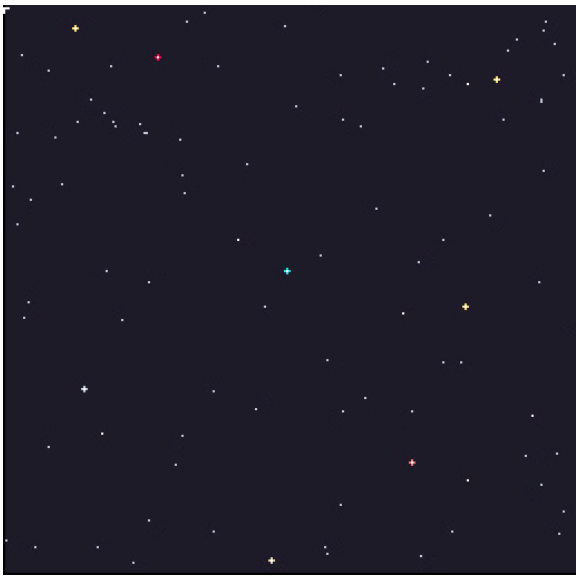


FIGURE 38 – Tile set permettant de faire un décord étoilé