Rapport de soutenance

PROJECT F.S

PROJECT F.S

Make The Great Adventure

Martin Molines

Thomas Ma-Paw-Youn

Gauthier Ruellan

Aurélien Soula

Mai 2025

Table des matières

1 Introduction **1**

2 Reprise du cahier des charges **2**
2.1 Introduction 2

Introduction **2**
2.2 Mécaniques de jeu / Gameplay 3
2.3 Graphismes 3
2.4 Musiques et Sons 4
2.5 Intelligence Artificielle 4
2.6 Multijoueur 4
2.7 Site Web 5

3 Début du Projet - Premier Prototype **6**
3.1 Mise en place du joueur et gestion des inputs 6
3.2 Intelligence Artificielle 7
3.3 Prototype de map 11
3.4 Prototype de système de tir 14

4 Avancement au mois de mars **20**
4.1 Amélioration de la physique 20
4.2 Mécaniques de jeu 21
4.2.1 Changement d'armes 21
4.2.2 Dash 22
4.2.3 Grappin 25
4.3 Map 26
4.4 Écran titre 31
4.5 Interface utilisateur 33
4.6 Amélioration des ennemis 34
4.7 Outils de debug 37
4.8 Site Web 37

5 État final **40**
5.1 Level-design et décoration 40
5.2 Nouveaux ennemis 41
5.3 Modèles 3D et animations 41
5.4 Finalisation des menus 43

5.5 Gestion des options 45
5.6 Animations des joueurs synchronisées en ligne 46
5.7 Gestion des événements et des ennemis 47
5.8 SFX et VFX 48
5.9 Site Web 49

6 Conclusion **52**

1 Introduction

Ce rapport de soutenance présente l'état final de notre projet de première année à l'EPITA, pour lequel nous avons réalisé un fast FPS, le jeu *Project F.S*.

Nous commençons par plusieurs rappels : de ce qui était prévu, initialement, dans le cahier des charges, tout d'abord, puis de ce qui était réalisé dans les deux soutenances intermédiaires.

Nous détaillons ensuite tous les ajouts effectués lors des deux derniers mois et mettons en évidence comment le jeu et son site web ressemblent maintenant à ce qui était prévu, autant pour la partie solo que pour la partie multijoueur.

2 Reprise du cahier des charges

2.1 Introduction

Ce cahier des charges présente le jeu intitulé "*Project F.S*", réalisé par l'entreprise MTGA dans le cadre de l'année scolaire 2024-2025 de l'EPITA, école d'ingénieurs informatique. Ce cahier présentera donc les objectifs et aspects techniques du projet (le jeu et son site internet) ainsi que les moyens utilisés. Certaines parties peuvent être amenées à changer, mais le cadre initial restera toujours le même, conservant la vision originale du projet.

Project F.S est un jeu vidéo de type *Fast FPS/Movement shooter*. Celui-ci sera principalement axé sur le gameplay et plus particulièrement sur les déplacements du personnage joueur dans l'environnement. L'histoire se déroule dans le cadre futuriste d'un laboratoire, plus particulièrement dans des chambres de tests – tests auxquels le joueur est forcé de participer.

Le contexte du jeu sera brièvement introduit par un texte défilant avant que le joueur commence l'aventure. Ce dernier sera ensuite directement envoyé dans la première chambre de tests dans laquelle les bases du jeu lui seront enseignées. Le protagoniste enchaînera donc les chambres de tests dans lesquelles il devra vaincre de nombreux ennemis avec les armes fournies par le laboratoire. Au fur et à mesure de sa progression, le joueur rencontrera des ennemis de plus en plus puissants et débloquera de nouvelles capacités. L'histoire restera volontairement simpliste pour laisser une plus grande place au gameplay, nécessitant une grande concentration de par sa vitesse et sa difficulté.

Les graphismes se rapprocheront du style des premières consoles de jeux 3D (Play Station 1, Play Station 2, Nintendo 64) qualifiés de "Low-Poly", c'est-à-dire des modèles 3D utilisant peu de polygones. Les décors ainsi que les couleurs seront également simplistes sans être répétitifs.

Ce projet va nous permettre d'évoluer sur plusieurs aspects : que ce soit sur le plan du travail d'équipe, primordial à la réussite du projet, ou celui de la conception d'un projet à plus grande échelle dans un temps limité, ce qui nous préparera au monde du travail.

2.2 Mécaniques de jeu / Gameplay

Pour ce qui est du Gameplay, le point de vue sera bien évidement à la première personne et les actions classiques de tous les FPS déja existants, c'est-à-dire se déplacer dans toutes les directions, regarder dans toutes les directions, sauter, tirer et changer d'arme, seront disponibles.

Malgré tout, il y a une différence majeure : les armes ne se rechargent pas et ont des munitions à l'infini pour permettre un gameplay plus fluide avec moins de temps de pause. Néanmoins, pour inciter le joueur à diversifier sa façon de jouer et à tester toutes les armes du jeu, un bonus de dégâts sur la première balle sera appliqué si le joueur change d'arme.

De plus, pour ce qui est des actions liées à la transversalité, le joueur disposera d'un double saut et pourra "wall-run" (courir sur les murs) et aura un grappin qui sera essentiel dans sa prise de hauteur.

La disposition de la plupart des cartes sera cubique. En effet, cette disposition exploite au mieux la transversalité et rentre aussi parfaitement dans le thème "salles de tests".

Le mode solo sera composé de quelques niveaux à la difficulté exponentielle et le mode multijoueur sera un mode versus qui vous fera affronter d'autres joueurs.

2.3 Graphismes

Pour les graphismes, la 3D est obligatoire car elle est nécessaire au genre du FPS. Pour l'aspect modélisation et animation 3D, nous allons utiliser les outils intégrés a Unity. Nous utiliserons également des packs de modèles et d'animations disponibles sur le marketplace d'Unity ou sur d'autres sources en ligne. Tous les modèles et assets utilisés seront évidemment en accord avec le style Low-Poly du jeu. Les cartes, les éléments du décors ainsi que les ennemis seront d'un aspect simple mais suffisamment variés et différentiables dans le but de pousser le gameplay en avant. Le *Fast FPS* étant un genre très rapide et nerveux, il est important que l'environnent soit facilement lisible par le joueur, d'où ces choix graphiques. Les graphismes n'étant pas le centre du jeu, ils serviront surtout de support au gameplay. Le choix des couleurs sera également en corrélation avec cette philosophie, tendant plus vers des couleurs primaires ainsi que le blanc.

2.4 Musiques et Sons

La musique et les bruitages jouent un rôle essentiel dans l'expérience d'un jeu vidéo. Tout d'abord, ils participent à la création d'une ambiance immersive, permettant au joueur de s'immerger pleinement dans l'univers du jeu. La bande sonore, qu'il s'agisse de musique ou d'effets sonores, contribue à renforcer les émotions et à accentuer les moments clés, que ce soit pour instaurer une atmosphère de tension, de suspense ou de détente.

Mais au-delà de l'immersion, le son a aussi un impact direct sur l'aspect tactique du jeu. L'ouïe devient un sens fondamental, notamment dans certains genres comme celui des jeux de tir à la première personne (FPS) que nous avons choisi pour ce projet. Dans ce type de jeu, les bruits de pas, les tirs, et même les rechargements d'armes fournissent des informations cruciales pour le joueur. Ces indices sonores peuvent déterminer une stratégie gagnante en permettant d'anticiper les mouvements de l'adversaire ou de repérer des ennemis cachés. Un bon joueur de FPS se fie non seulement à sa vision, mais aussi à son écoute attentive pour réagir au bon moment.

2.5 Intelligence Artificielle

L'utilisation d'une forme d'intelligence artificielle sera omniprésente dans le jeu, car elle sera utilisée pour déterminer les actions des nombreux ennemis qui apparaîtront dans les différents niveaux. Ces actions ne découleront pas de réseaux de neurones apprenants (ce qui dépasserait le cadre de ce projet) mais ne seront pas non plus, à l'autre extrême, que le produit d'une suite d'instructions prédéfinies. On pourra réellement parler d'intelligence artificielle puisque les ennemis choisiront comment se déplacer et attaquer en tenant compte de paramètres tels que la position du personnage joueur. Pour accomplir cette tâche, les fonctionnalités de *pathfinding* pré-implémentées dans Unity seront utilisées. Les différents types d'ennemis ne réagiront pas de la même façon aux informations récupérées. Par exemple, certains pourraient tenter de se rapprocher du protagoniste pour l'attaquer au corps à corps tandis que d'autres s'éloigneraient pour lancer des attaques à distance.

2.6 Multijoueur

Dans *Project F.S* le multijoueur se déroulera sous la forme d'un mode de jeu spécifique en Joueur contre Joueur (PvP). Le but de chaque joueur sera de

réussir à éliminer le plus de joueurs adverses dans un temps imparti. Toutes les mécaniques de gameplay de base du mode solo seront aussi disponibles dans le mode multijoueur. D'un point de vue plus technique, la mise en place de ce mode se fera par l'utilisation d'un WAN (*Wide Area Network*) couplé à des outils intégrés à Unity ainsi qu'à l'API PUN (*Photon Unity Networking*). Lors de la création d'une partie multijoueur, une personne sera définie comme "hôte" et les autres joueurs pourront se connecter directement à sa partie grâce aux technologies précédemment citées.

2.7 Site Web

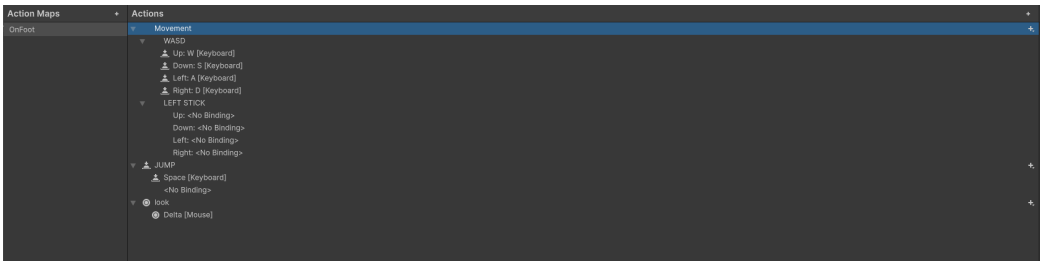
Le site web contiendra les informations relatives au projet comme la présentation de l'équipe de développement, une brève description du jeu avec des instructions par exemple pour le mode multijoueur, un lien de téléchargement ainsi que des liens vers les ressources utilisées. Il sera hébergé sur Github Pages. Sa charte graphique sera logiquement en accord avec celle du jeu.

3 Début du Projet - Premier Prototype

Ici sont décrits les divers éléments que nous avons intégrés dans le prototype ainsi que les diverses pistes d'améliorations que nous avons trouvées.

3.1 Mise en place du joueur et gestion des inputs

Nous avons donc commencé (en particulier Gauthier) par la mise en place d'un personnage joueur (qui n'est pour le moment qu'une capsule). C'est avec cette dite capsule que nous avons testé et utilisé le Unity Input System. Il s'agit d'un outil intégré à Unity qui permet de prendre en charge différents périphériques et de facilement les paramétrer pour les implémenter dans un projet. Nous nous en sommes donc servi pour faire les mouvements de base du personnage joueur, c'est-à-dire marcher.



Pour faire fonctionner le tout, nous avons aussi dû écrire plusieurs scripts C#. Le premier est l'InputManager qui est chargé de détecter les inputs du joueur et de les utiliser dans les fonctions adéquates, comme se déplacer ou sauter, par exemple.

```
public class InputManager : MonoBehaviour
{
    private PlayerInput playerInput;
    private PlayerInput.OnFootActions onFoot;

    private PlayerMotor motor;

    private look look;

    // Start is called once before the first execution of Update after the MonoBehaviour is created
    Event function  Gauthier Ruellan
    void Awake()
    {
        playerInput = new PlayerInput();
        onFoot = playerInput.OnFoot;
        motor = GetComponent<PlayerMotor>();
        onFoot.JUMP.performed += ctx:CallbackContext => motor.Jump();
        look = GetComponent<look>();
    }
```

Toutes les modifications de mouvement sont elles faites dans un script à part entière qui reçoit les mouvements de l'InputManager et les exécute en conséquence. La plus basique de ces exécutions est donc la marche, qui consiste simplement à déplacer le joueur sur les axes x et z.

```
public void ProcessMove(Vector2 input)
{
    Vector3 MoveDirection = Vector3.zero;
    MoveDirection.x = input.x;
    MoveDirection.z = input.y;
    controller.Move(motion:transform.TransformDirection(MoveDirection) * speed * Time.deltaTime);

    playervelocity.y += gravity * Time.deltaTime;
    if (isgrounded && playervelocity.y < 0)
    {
        playervelocity.y = -2f;
    }
    controller.Move(motion:playervelocity * Time.deltaTime);

}
```

3.2 Intelligence Artificielle

Pour implémenter des ennemis "intelligents", nous avons commencé par consulter un tutoriel sur l'implémentation de l'IA pour un jeu en 3 dimensions sur Unity, nous apprenant à utiliser les fonctions pré-implémentées de mouvement des IA. Cette vidéo utilisait trois états pour les ennemis : en cours de patrouille,

de poursuite et d'attaque, l'ennemi changeant d'état selon la position du joueur. Nous n'avons pour le moment implémenté que les deux premiers.

```
void Awake()
{
    player = GameObject.FindGameObjectWithTag("Player").transform;
    agent = GetComponent<NavMeshAgent>();
}

// Update is called once per frame
Event function   Aurelien Soula
void Update()
{
    playerInSightRange = Physics.CheckSphere(transform.position, sightRange, whatIsPlayer);

    if (!playerInSightRange) Patroling();
    else ChasePlayer();
}
```

Codes des fonctions Awake, permettant d'initialiser les attributs "player" (pour localiser le joueur) et "agent" (pour accéder à son *NavMesh agent*) de l'ennemi, et Update, la fonction qui actualise son action en permanence

Pour l'état de patrouille, l'ennemi prend un point au hasard dans sa distance de recherche, vérifie si ce point est atteignable et si oui, s'y dirige grâce à son *NavMesh agent*. Une fois le point atteint, l'ennemi cherche un nouveau point de la même façon.

```
private void Patroling()
{
    if (!walkPointSet) SearchWalkPoint();

    if (walkPointSet)
    {
        agent.SetDestination(walkPoint);
    }

    Vector3 distanceToWalkPoint = transform.position - walkPoint;

    if (distanceToWalkPoint.magnitude < 1f)
    {
        walkPointSet = false;
    }
}

Frequently called   1 usage   Aurelien Soula
private void SearchWalkPoint()
{
    float randomZ = Random.Range(-walkPointRange, walkPointRange);
    float randomX = Random.Range(-walkPointRange, walkPointRange);


    walkPoint = new Vector3(transform.position.x + randomX, transform.position.y, transform.position.z + randomZ);

    if (Physics.Raycast( origin: walkPoint, direction: -transform.up, maxDistance: 2f, whatIsGround)) walkPointSet = true;
}
```

Code de la fonction de patrouille et de sa fonction auxiliaire SearchWalkPoint

La fonction de chasse, elle, est beaucoup plus simple. Il nous suffit de prendre la position du joueur comme destination et le *NavMesh agent* se charge du reste.

```
private void ChasePlayer()
{
    agent.SetDestination(player.position);
}
```

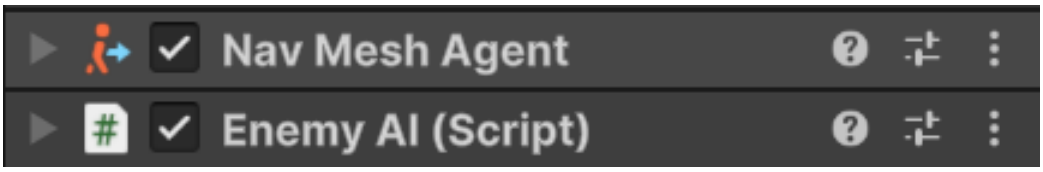
Code de la fonction de chasse

Les ennemis possèdent également une fonction TakeDamage, qui est appelée lorsqu'ils sont touchés par les attaques du personnage joueur et qui décrémente simplement l'attribut "Health" de l'ennemi considéré.

```
public void TakeDamage(int damage)
{
    Health -= damage;
}
```

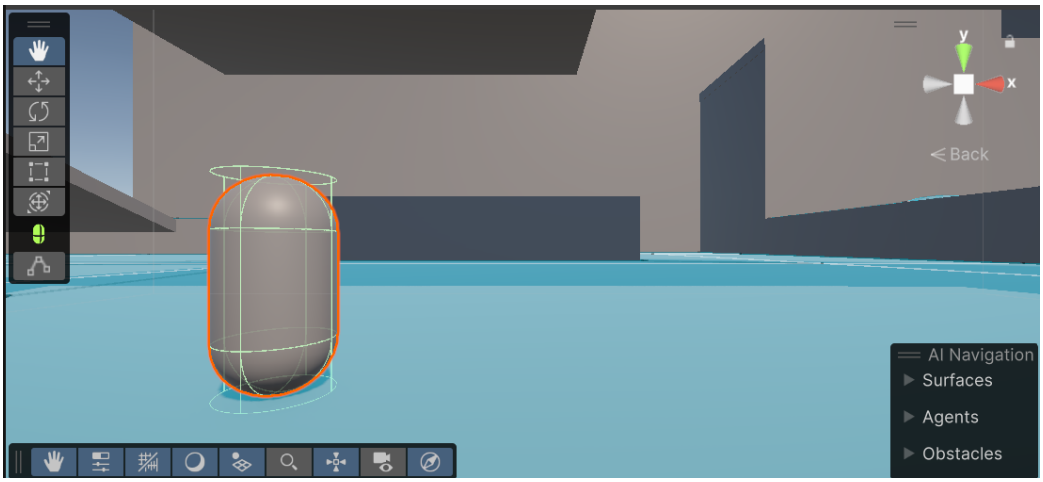
Code de la fonction TakeDamage

Une fois le script fini, nous l'avons ajouté à notre prototype d'ennemi. Nous lui avons également ajouté le fameux *NavMesh agent* qui gère ses déplacements.



Pour que ce script fonctionne correctement, nous avons également eu à ajouter des "layers", plus précisément le layer whatIsPlayer sur lequel se trouve le protagoniste et le layer whatIsGround où sont placés les objets sur lesquels les ennemis peuvent marcher.

Pour finir, nous avons ajouté une *NavMesh surface* pour que le *NavMesh agent* sache quelles surfaces utiliser pour calculer le chemin à prendre.

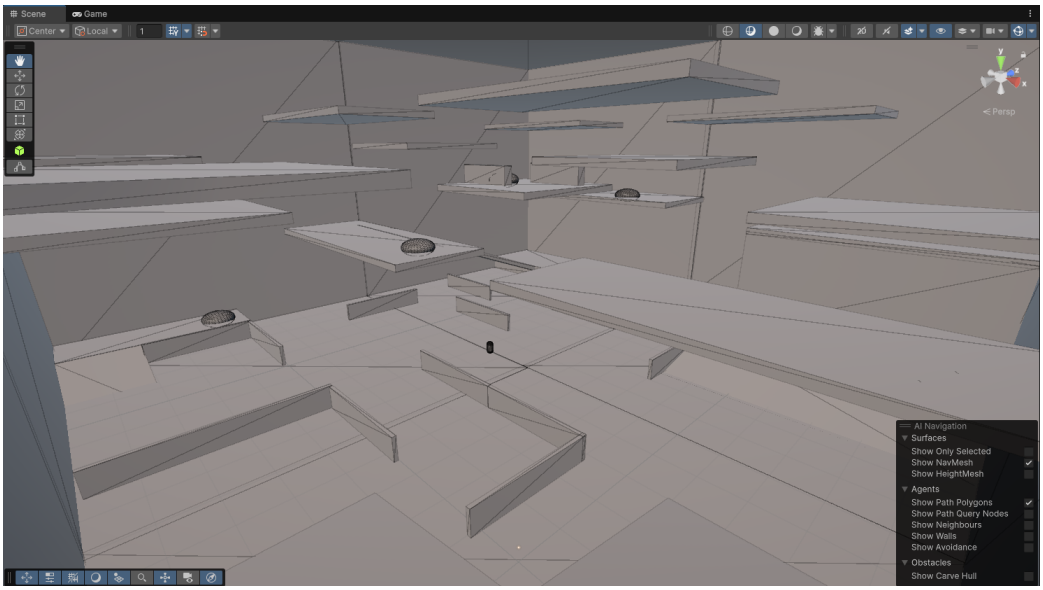


En bleu, les zones sur lesquels la capsule ennemie peut se déplacer

Nous avons donc implémenté une IA pour un ennemi prototype qui se déplace vers le joueur quand celui-ci entre dans son champ de vision. Cette partie a été gérée en majorité par Aurélien.

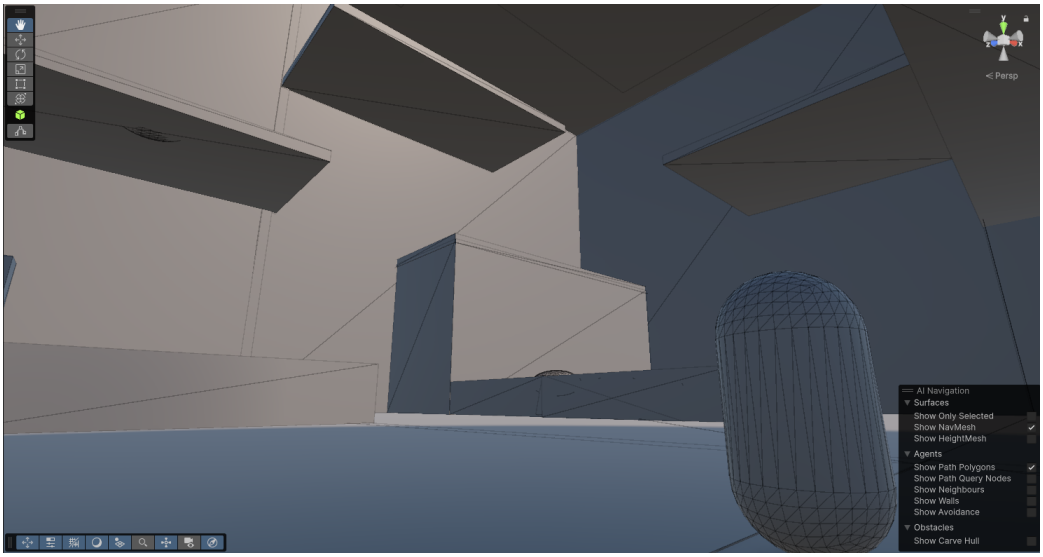
3.3 Prototype de map

Nous avons établi une première maquette de la première carte de Project F.S :

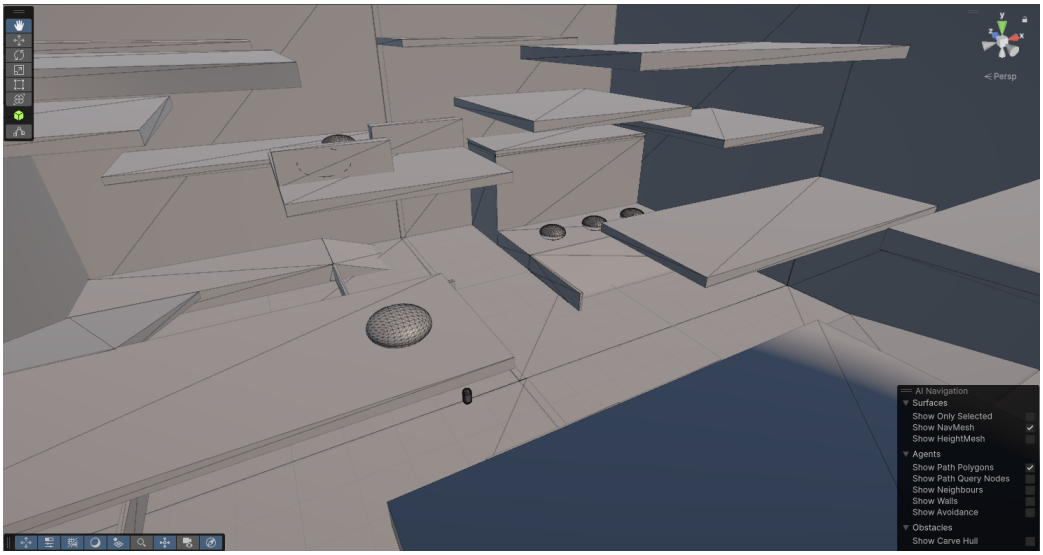


Comme nous pouvons le voir sur cette capture d'écran, le level design se dispose sur plusieurs niveaux. En effet, le joueur (ici représenté par la petite capsule au milieu de l'image) pourra se déplacer où bon lui semble grâce à ses différentes capacités de mobilité. La carte a donc été pensée de manière à fluidifier les mouvements du joueur en lui permettant de s'élever facilement vers les étages supérieurs.

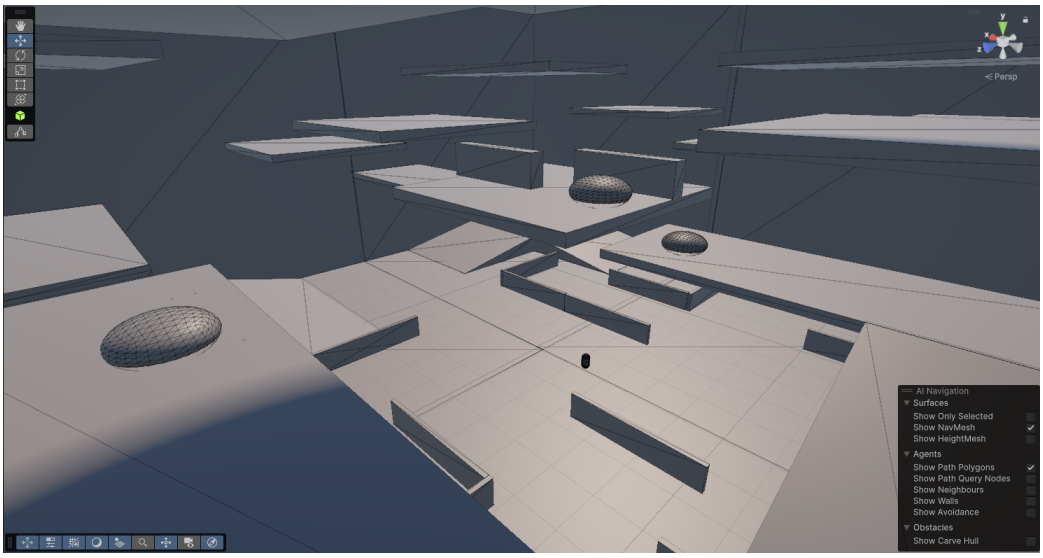
Pour vous donner un ordre de grandeur :



Le joueur se situe donc dans une énorme pièce cubique, constituée de murs lui permettant de se mettre à couvert en cas de difficulté, et de différentes plateformes lui permettant de prendre beaucoup de hauteur. Certains endroits de la carte sont pensés comme étant relativement "plus sûrs", et dans lesquels des "kits de soins" seront mis à disposition pour permettre au joueur de regagner rapidement des points de vie et de reprendre ensuite le combat. D'autres de ces espaces sont des points de réapparition pour le mode multijoueur, ce qui permettra aux joueurs de ne pas tout le temps réapparaître au même endroit et donc d'empêcher une mort instantanée lors de la réapparition.



Les formes sphériques que nous pouvons apercevoir représentent des "tremplins" et serviront à propulser le joueur pour lui donner encore plus de liberté de mouvement.



Thomas est le contributeur principal à la map.

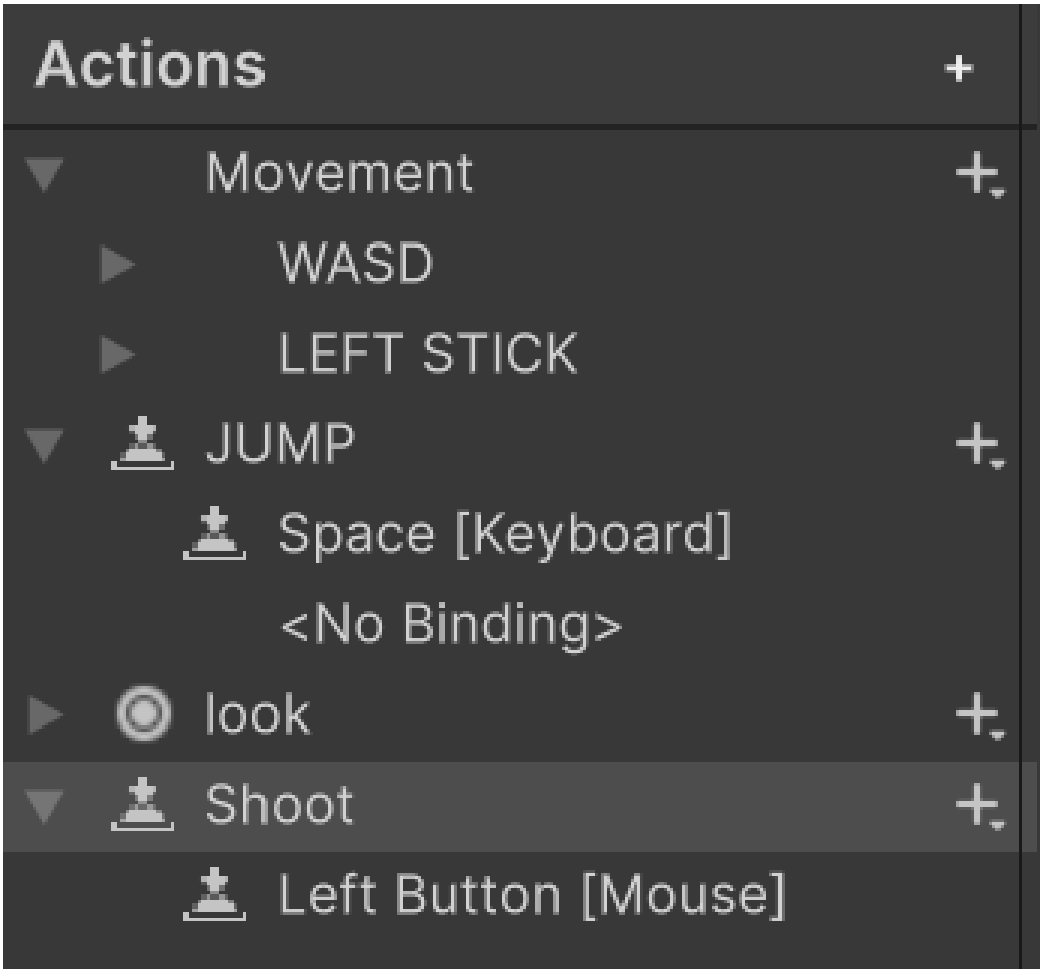
3.4 Prototype de système de tir

Afin de mettre en place notre système de tir, réalisé par Aurélien, nous avons, pour cette fonctionnalité également, été aidés par un tutoriel vidéo. Cependant, nous avons effectué plusieurs changements pour que l'implémentation de cet aspect crucial de notre jeu soit conforme à notre vision.

Nous avons créé la nouvelle classe GunSystem dans laquelle nous avons commencé par implémenter le même système de gestion d'input que notre modèle, mais, ayant déjà un "input system", nous savions que cette fonction allait être temporaire.

```
private void InputGet()
{
    if (allowButtonHold)
    {
        shooting = Input.GetMouseButton(0);
    }
    else
    {
        shooting = Input.GetMouseButtonDown(0);
    }

    if (readyToFire && shooting)
    {
        bulletsShot = bulletsPerShot;
        Shoot();
    }

}
```

Des modifications ont plus tard été apportées à notre "input system" pour prendre en compte l'action de tirer, remplaçant cette fonction.


Actions
Movement
WASD
LEFT STICK
JUMP
Space [Keyboard]
<No Binding>
look
Shoot
Left Button [Mouse]

```
gun = GetComponentInChildren<GunSystem>();
onFoot.Shoot.performed += StartShooting;
onFoot.Shoot.canceled += StopShooting;
```

nouvelles lignes de la fonction Awake de InputManager.cs

```
private void StartShooting(InputAction.CallbackContext context)
{
    gun.shooting = true;
}

1 usage   Aurelien Soula
private void StopShooting(InputAction.CallbackContext context)
{
    gun.shooting = false;
}
```

Fonctions ajoutées à InputManager.cs

Nous avons également implémenté la fonction Shoot, qui utilise la fonction Physics.Raycast pour trouver le point d'impact. Elle gère à la fois le retour visuel et le fait d'infliger des dégâts aux ennemis.

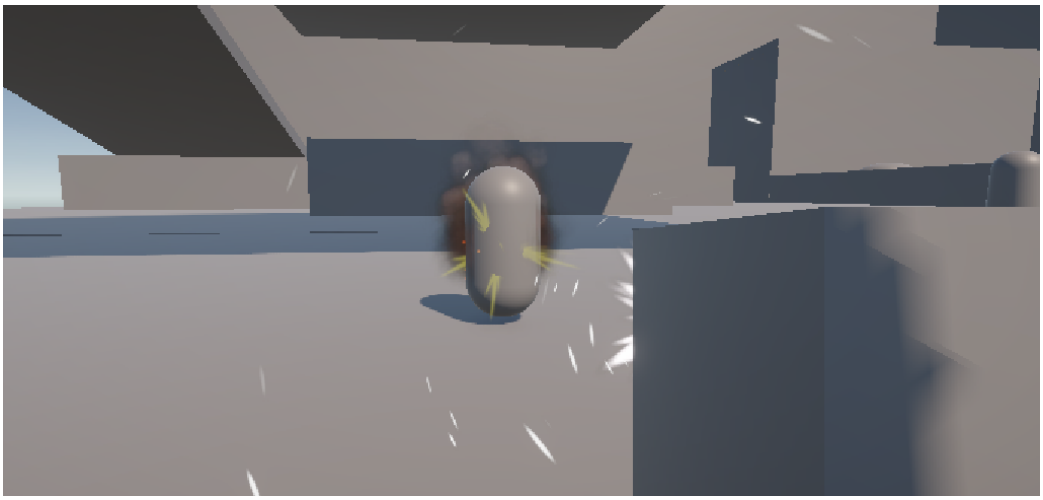
```
public void Shoot()
{
    readyToFire = false;

    if (Physics.Raycast( origin: cam.transform.position,  direction: cam.transform.forward, out RayHit, range))
    {
        Debug.Log(RayHit.collider.name);

        if (RayHit.collider.CompareTag("Enemy"))
        {
            RayHit.collider.GetComponent<EnemyAI>().TakeDamage(damage);
        }
    }



    Instantiate(bulletHoleGraphic, RayHit.point,  rotation: Quaternion.Euler(0, 180, 0));
    Instantiate(muzzleFlash,attackPoint.position, Quaternion.identity);

    bulletsShot--;

    Invoke(nameof(ResetShot),timeBetweenShooting);

    if (bulletsShot > 0)
    {
        Invoke( methodName: "Shoot",timeBetweenShots);
    }
}
```

```
private void ResetShot()
{
    readyToFire = true;
}
```

Code de la fonction Shoot et de sa fonction auxiliaire ResetShot. Ce code prévoit de tirer plusieurs balles en un appui sur la touche mais nous n'avons pas encore implémenté d'armes utilisant cette option

Après des difficultés pour trouver des assets graphiques permettant un bon retour visuel, nous sommes finalement arrivés à une première version de notre

système de tir.



Prototype du système de tir, avec un cube en guise d'arme

4 Avancement au mois de mars

4.1 Amélioration de la physique

Nous avons amélioré le moteur physique qui avait déjà été précédemment montré. Ceci a principalement été réalisé par Gauthier avec l'aide d'Aurélien. Tout d'abord, nous avons rajouté la conservation de vélocité lors des sauts. Si le joueur court avant de sauter, alors la trajectoire du saut sera modifiée. Ce changement est appliqué à tout objet modifiant la vélocité du joueur.

```
else
{
    controller.Move(motion: transform.TransformDirection(MoveDirection) * speed * Time.deltaTime);
    direction = MoveDirection;
    playervelocity.x = MoveDirection.x;
    playervelocity.z = MoveDirection.z;
}
```

Code qui calcule la vélocité aérienne du joueur et modifie sa trajectoire

De plus, nous avons rajouté la possibilité d'annuler cette vélocité en appuyant sur la touche dans le sens inverse du mouvement. Cela aura pour effet de stopper le joueur dans les airs, il ne pourra donc plus que légèrement influer sur sa trajectoire de chute.

```
if (math.abs(MoveDirection.z) < math.abs(playervelocity.z))
{
    playervelocity.z = 0;
    MoveDirection.z = 0;
}
if (math.abs(MoveDirection.x) < math.abs(playervelocity.x))
{
    playervelocity.x = 0;
    MoveDirection.x = 0;
}
```

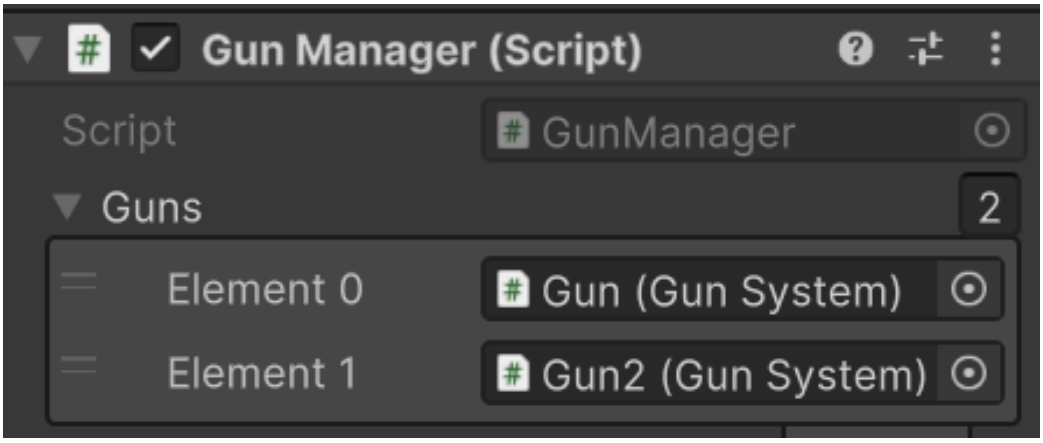
La partie de la fonction qui annule la vélocité aérienne

Enfin, il y a eu de nombreuses corrections de bugs et optimisations, notamment concernant la gestion de la gravité. Nous avons fait le choix de séparer le vecteur de gravité du personnage joueur de son vecteur de déplacement principal. En effet, les deux rentraient parfois en conflit et cela entraînait un déplacement diagonal du personnage.

4.2 Mécaniques de jeu

4.2.1 Changement d'armes

Nous avons implémenté le changement d'armes dans le jeu. Cette tâche a été majoritairement réalisée par Aurélien. L'objet représentant le protagoniste compte désormais parmi ses "enfants" un objet avec le script GunManager. Ce script possède une liste d'objets de type GunSystem qui peut être parcourue avec le clic droit de la souris. Un seul objet de la liste est actif à un instant donné.



Le GunManager

```
public void ChangeGun()
{
    _gunIndex = (_gunIndex + 1) % Guns.Count;
    CurrentGun.shooting = false;
    CurrentGun.gameObject.SetActive(false);
    CurrentGun = Guns[_gunIndex];
    CurrentGun.gameObject.SetActive(true);
}
```

La fonction ChangeGun de GunManager.cs

La fonction ChangeGun est appelée par notre InputManager. Au départ, nous n'avions pas la ligne qui faisait que l'arme s'arrête de tirer avant d'être échangée. Mais sans elle, si l'on changeait d'arme tout en tirant, au moment de revenir sur l'arme qui était en train de tirer, elle continuait à tirer indéfiniment, ce qui ne nous paraissait pas souhaitable.

Nous n'avons actuellement que deux armes mais le système est fait pour être extensible.

4.2.2 Dash

Nous avons implémenté la possibilité de "dash" (se propulser) dans le jeu. L'entièreté de cette mécanique du jeu a été réalisée par Gauthier. Tout d'abord, nous avons dû créer une fonction avec une structure **coroutine**, c'est-à-dire une fonction principale qui en appelle une autre qui dépend du passage du temps dans le jeu. Le dash ne pouvant pas être utilisé continuellement, il y a un temps d'attente de 0,5 seconde entre deux dash. Pour éviter les problèmes de calculs du temps sur des machines moins performantes ou pouvant être causés par des bugs, la **coroutine** permet de calculer le temps dans le jeu. Peu importe à quelle vitesse tourne le jeu, le temps entre chaque dash restera le même. De plus, pour éviter que le joueur puisse rester dans les airs indéfiniment, nous avons implémenté une limite de deux dash aériens avec obligation de toucher le sol pour en récupérer.

```
1 usage   Gauthier Ruellan +1
public void dash()
{
    if (dashcounter>0 && OffCdDash)
    {
        StartCoroutine( routine: DashCoroutine());
        dashcounter--;
        dashTimer = dashCooldown;
    }

}
```

La fonction dash principal appelant la coroutine

```
private IEnumerator DashCoroutine()
{
    var MoveDirection :Vector3 = Vector3.zero;
    MoveDirection = direction.normalized;

    if (MoveDirection.x ==0 & MoveDirection.z == 0)
    {
        MoveDirection.z = 20;
    }
    else
    {
        MoveDirection.z *= 20;
        MoveDirection.x *= 20;
    }
```

Partie de la coroutine responsable du calcul de la trajectoire

Pour calculer la trajectoire du dash, nous utilisons la valeur "direction" que nous normalisons. Si *direction* est nulle alors le dash sera par défaut dans le sens de la caméra du joueur, sinon elle suivra les commandes du joueur comme pour un déplacement normal.

```
float startTime = Time.time;
while(Time.time < startTime + dashTime)
{
    controller.Move( motion: transform.TransformDirection(MoveDirection) * dashSpeed * Time.deltaTime)
    Gravity2.y = 0;

    yield return null;
}
```

Partie de la coroutine responsable de la détection du temps et du déplacement

Cette partie présente l'intérêt principal de la coroutine. Elle permet d'exécuter les instructions dans l'intervalle de temps de durée "dashTime" qui commence

logiquement au moment où l'action est lancée. Pendant ce temps, le joueur sera donc déplacé pour réaliser le dash.

4.2.3 Grappin

Un système de grappin a été implémenté par Martin afin de rajouter une mécanique de déplacement et d'augmenter la mobilité du personnage. Ce système permet au joueur de s'agripper à certains murs ou objets en visant ce dernier et en appuyant sur la touche dédiée. Lorsque le joueur active le grappin, un raycast est lancé depuis sa position jusqu'à l'endroit regardé et, si ce dernier est un objet compatible, alors un mouvement associé est activé avec une coroutine pour se déplacer progressivement vers le point d'arrivée. Le déplacement est effectué selon un vecteur directionnel normalisé multiplié un coefficient (la variable GrabSpeed). Une vérification de distance vérifie si le déplacement devient incohérent et l'arrête instantanément le cas échéant.

```
private IEnumerator GrappinCoroutine(Vector3 donnée, Collider hitObject)
{
    Physics.IgnoreLayerCollision(6,11);

    Vector3 directionmieux = (donnée - transform.position).normalized;
    _gravity2.y = 0;

    while(Vector3.Distance(transform.position, donnée)>1f)
    {
        Vector3 tideplacement = (directionmieux * GrabSpeed * 50 * Time.deltaTime);
        if (Vector3.Distance(transform.position + tideplacement, donnée) < 1f)
        {
            break;
        }
        if (Vector3.Distance(transform.position + tideplacement, donnée) > Vector3.Distance(transform.position,donnée))
        {
            break;
        }


        _controller.Move(tideplacement);
        yield return null;
    }
```

Fonction principale du grappin

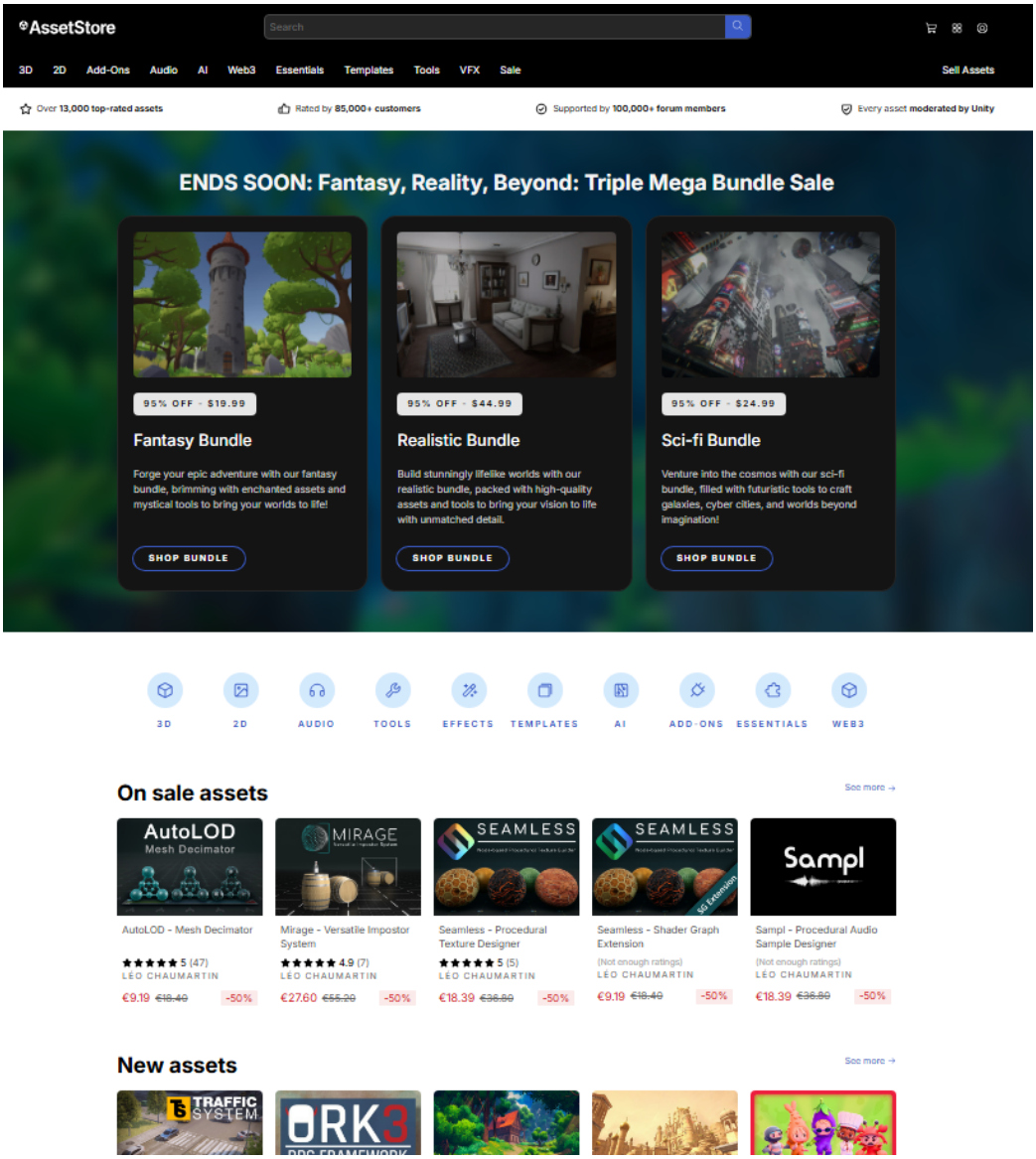
Il existe également des objets de type GrappinBoost, qui sont des orbes qui ont pour fonction de donner un boost de vitesse après les avoir atteints en utilisant le grappin. Cela rajoute une dimension acrobatique à notre jeu.

```
if (hitObject.CompareTag("GrappinBoost"))
{
    _playervelocity += new Vector3(0, 0, 2);
    _gravity2.y = Mathf.Sqrt(f _jumpHeight * -3f * _gravity);
}
```

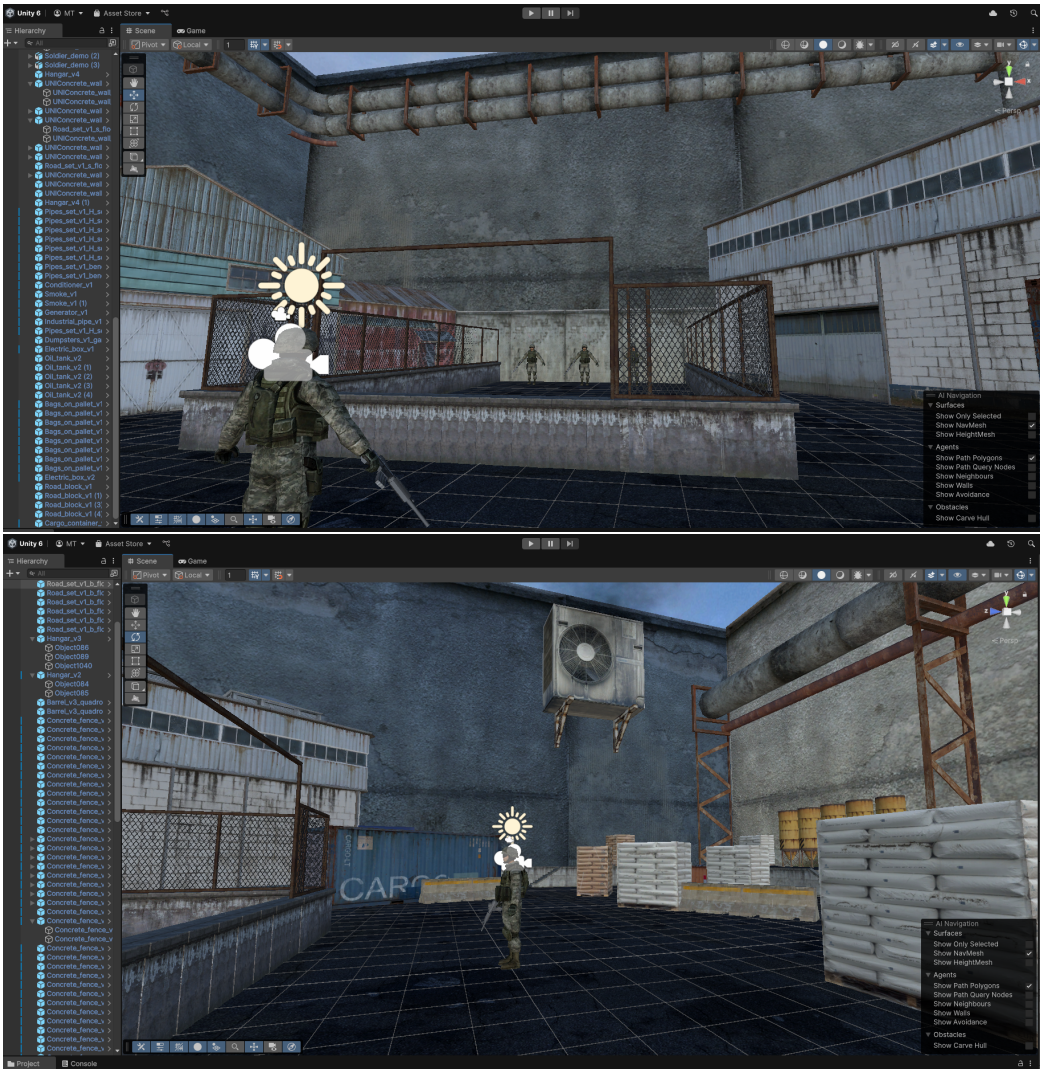
Partie de la fonction responsable du Boost

4.3 Map

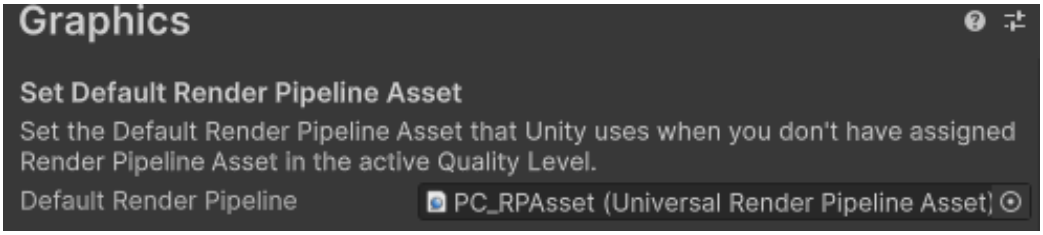
Pour ce qui est de l'environnement de jeu, nous avons choisi des premiers assets de textures et d'objets en tant que "placeholders" pour pouvoir commencer à imaginer le level design, mais aussi parce que nous nous sommes rendu compte d'un problème sur lequel nous reviendrons un peu plus tard. Nous avons trouvé ces assets sur le *Unity Asset Store*, dans lequel des milliers d'assets, créés par des professionnels et des indépendants, sont mis à disposition. Le projet étant financé par nous-mêmes, nous ne disposons pas du budget pour pouvoir nous fournir en assets payants. Nos choix se feront donc parmi les assets gratuits du site.



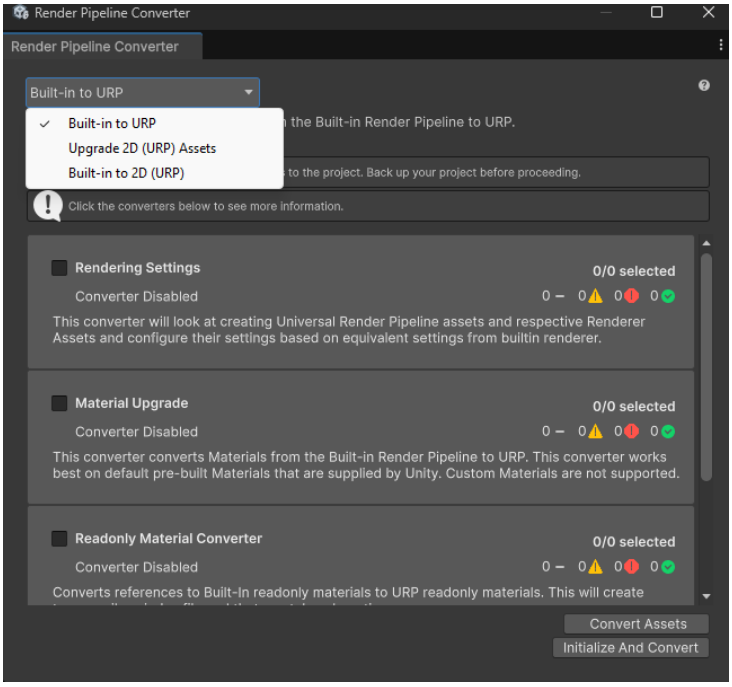
Voici un prototype de map que nous avons imaginé comme "lobby" ou "sandbox" pour que le joueur puisse tester les armes, ou tout simplement, s'amuser seul :



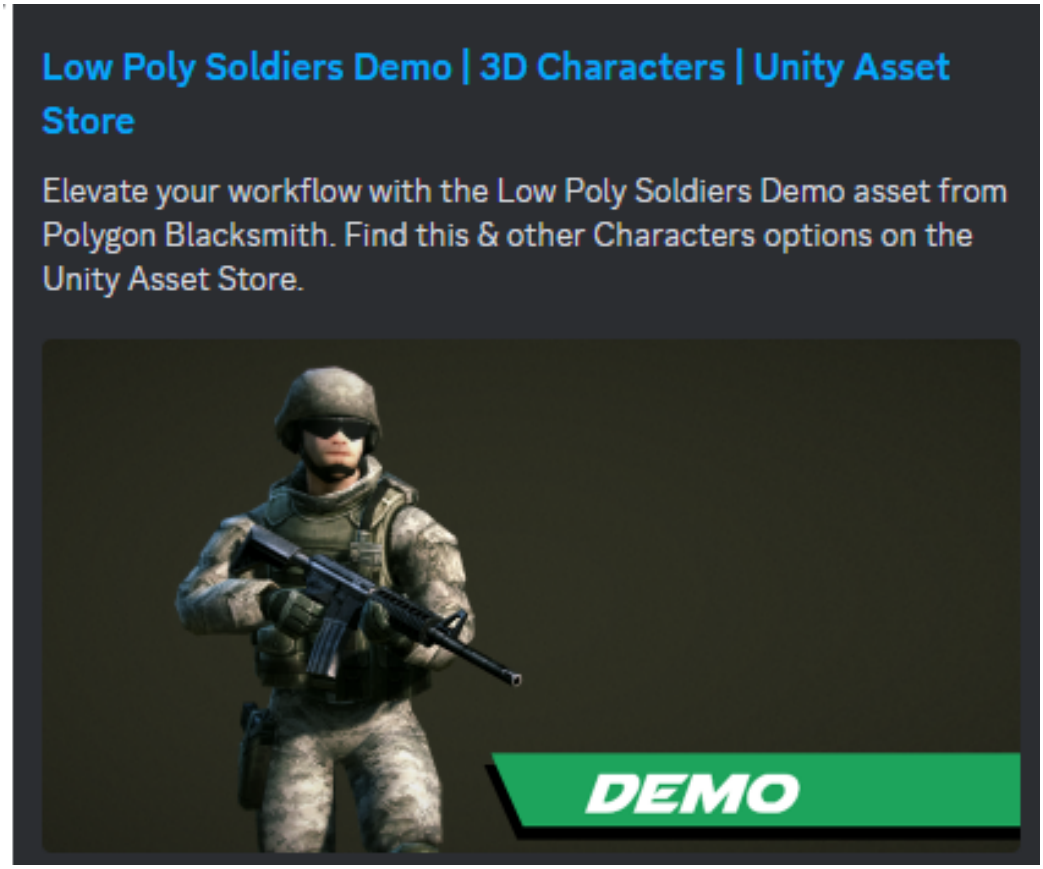
Comme vous pouvez le voir sur les screenshots, la direction artistique est bien différente de celle annoncée au départ. Cela est dû au problème esquissé précédemment : Unity dispose de plusieurs rendering pipelines, tels que URP (Universal Rendering Pipeline), Built-in Pipeline, etc. Au départ, par méconnaissance du sujet, nous avions décidé d'utiliser URP. Le problème étant qu'URP est une pipeline récente et sert surtout à optimiser des rendus faits sur plusieurs plateformes, telles que le mobile et la console. Or, Projet F.S. sera exclusivement sur PC, et les assets gratuits compatibles avec URP sont rares et pas forcément en lien avec la direction artistique du jeu.

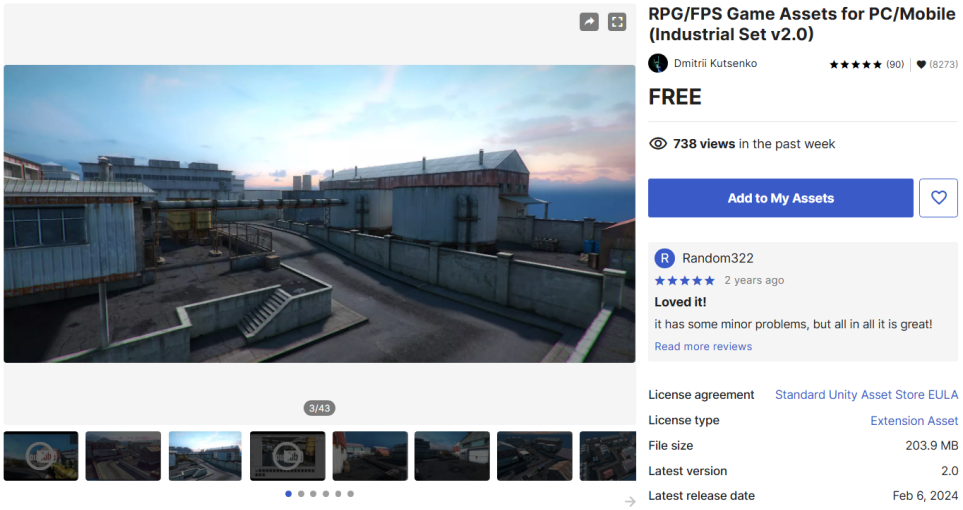


Nous avons donc décidé d'utiliser Built-in comme rendering pipeline, qui, comme le dit le nom, est la pipeline de base de Unity et est surtout utilisée pour les projets sur PC. Le problème étant qu'il existe un convertisseur automatique directement dans Unity pour convertir du Built-in vers URP, mais pas l'inverse. Ce qui veut dire qu'il va falloir convertir manuellement chaque asset et texture importés. Pour pouvoir rendre un projet fonctionnel, nous avons pour l'instant tout laissé en URP, mais cela sera notre premier changement une fois ce rendu intermédiaire passé.



Voici quelques assets choisis comme placeholders :





Partie réalisée par THOMAS MA-PAW-YOUN

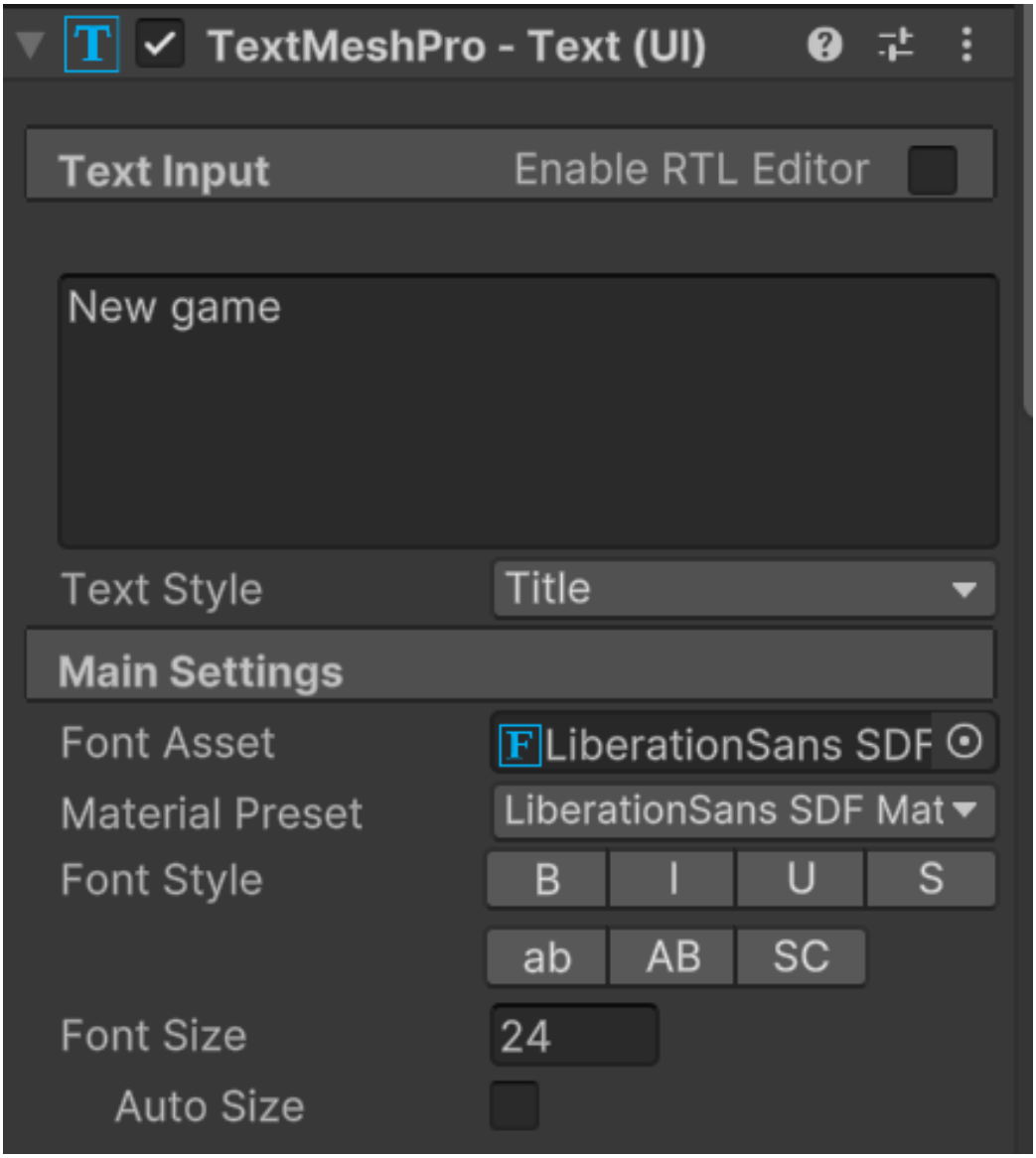
4.4 Écran titre

Pour réaliser l'écran titre, nous avons dû apprendre à utiliser un "canvas" pour ajouter une image et un bouton cliquable à une scène. Nous avons également utilisé TextMeshPro pour gérer le texte sur le bouton. Nous avons ensuite créé un nouveau script avec une fonction qui utilise LoadScene pour changer la scène.

L'écran titre

```
public void StartGame()
{
    SceneManager.LoadScene("demo");
}
```

La fonction appelée par le bouton

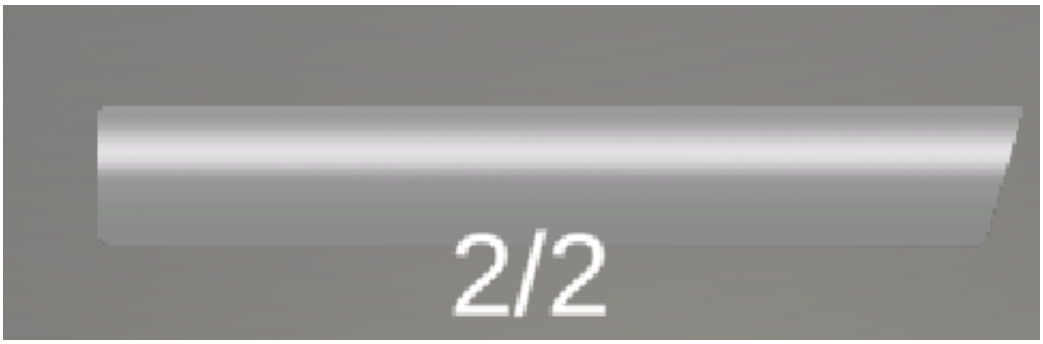


Le composant TextMeshPro

L'écran titre a été réalisé par Aurélien. Cette scène sera améliorée et complétée ultérieurement.

4.5 Interface utilisateur

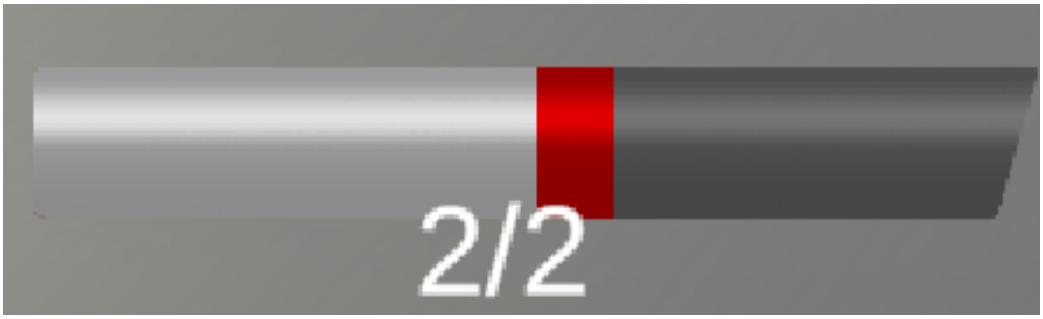
Nous avons mis en place une barre de vie et un compteur du nombre de dash disponibles, situés dans la partie inférieure de l'écran et visibles à tout moment.



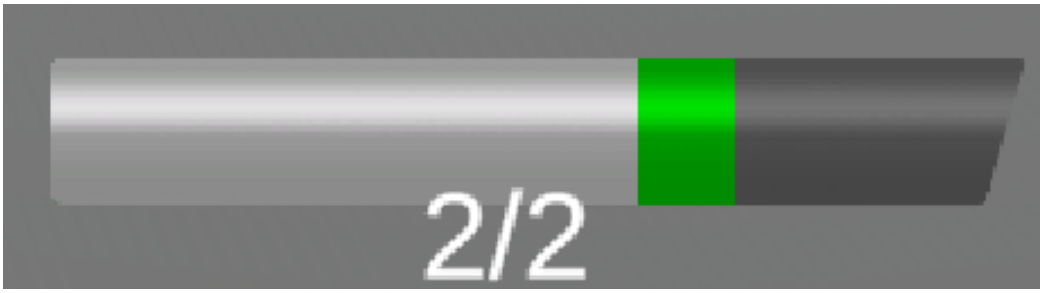
La barre de vie et le compteur de dash

Le compteur de dash est actuellement un simple texte TextMeshPro qui est actualisé quand le nombre de dash change. Cela pourrait changer par la suite.

La barre de vie est également animée et réagit fluidement aux dégâts et aux soins reçus. Cette animation est obtenue grâce à l'utilisation de trois images : une image de fond statique et deux barres superposées dont le remplissage est modifié. Lorsque des points de vie sont perdus, la barre de devant se "déplace" instantanément vers la gauche et la deuxième la suit progressivement grâce à une fonction d'interpolation linéaire (lerp). Quand des points de vie sont gagnés, ces rôles sont inversés. La barre "de derrière" change également de couleur selon la variation des points de vie.



Animation de dégâts de la barre de vie



Animation de soin de la barre de vie

```
public void UpdateHealthUI()
{
    float fillF = frontHealthBar.fillAmount;
    float fillB = backHealthBar.fillAmount;
    float hFraction = Health / _maxHealth;

    if (fillB > hFraction)
    {
        frontHealthBar.fillAmount = hFraction;
        backHealthBar.color = Color.red ;
        _lerpTimer += Time.deltaTime;
        float percentComplete = _lerpTimer / _chipSpeed;
        percentComplete = percentComplete * percentComplete;
        backHealthBar.fillAmount = Mathf.Lerp(fillB, hFraction, percentComplete);
    }

    if (fillF < hFraction)
    {
        backHealthBar.fillAmount = hFraction;
        backHealthBar.color = Color.green ;
        _lerpTimer += Time.deltaTime;
        float percentComplete = _lerpTimer / _chipSpeed;
        percentComplete = percentComplete * percentComplete;
        frontHealthBar.fillAmount = Mathf.Lerp(fillF, backHealthBar.fillAmount, percentComplete);
    }
}
```

Fonction qui gère le visuel de la barre de vie

Cet ajout a été réalisé majoritairement par Aurélien.

4.6 Amélioration des ennemis

Deux améliorations majeures ont été apportées aux ennemis. En plus de se déplacer, ils peuvent désormais attaquer et être détruits. Le processus de des-

truction est simple : quand l'ennemi prend des dégâts, nous vérifions si ses points de vie sont inférieurs à 0. Si c'est le cas, les composants non visuels de l'ennemi sont désactivés, il change de couleur et une animation de destruction est jouée. Son *gameObject* est finalement détruit après un court instant.

```
Frequently called   1 usage   Aurelien Soula
public void DestroyEnemy()
{
    Destroy(gameObject, t: 0.5f);
    foreach (var comp :Component in gameObject.GetComponents<Component>())
    {
        if (comp is not Renderer && comp is Behaviour behaviour)
        {
            behaviour.enabled = false;
        }
    }
    var destroyAnim :GameObject = Instantiate(_destroyAnim,gameObject.transform.position, Quaternion.identity);
    Renderer renderer = GetComponent<Renderer>();
    renderer.material.SetColor( nameID: BaseColor,Color.red );
    Destroy(destroyAnim, t: 1f);
}

Frequently called   1 usage   Aurelien Soula
public void TakeDamage(int damage)
{
    Health -= damage;
    if (Health <= 0)
    {
        DestroyEnemy();
    }
}
```

Fonction DestroyEnemy ainsi que la nouvelle fonction TakeDamage des ennemis

La mise en place des attaques fut plus complexe. Nous avons déterminé que, contrairement à ce que nous avons fait pour le protagoniste, il ne serait pas souhaitable d'utiliser le Ray Casting pour les tirs des ennemis car ils seraient alors trop difficiles à éviter. Nous avons donc fait le choix d'utiliser des projectiles, ce qui a nécessité l'ajout d'une nouvelle classe, Projectiles.

```
private void OnCollisionEnter(Collision collision)
{
    _collisions++;

    if (collision.collider.CompareTag("Player") && explodeOnTouch)
    {
        Explode();
    }
}

Frequently called   2 usages   Aurelien Soula *
private void Explode()
{
    if (explosion is not null)
    {
        var exp :GameObject = Instantiate(explosion, transform.position, rotation: quaternion.identity);
        Destroy(exp, t: 1f);
    }

    Collider[] targets = Physics.OverlapSphere(transform.position,explosionRange, whatIsEnnemy);
    foreach (var target :Collider in targets)
    {
        target.GetComponent<PlayerHealth>().TakeDamage(explosionDamage);
    }
    Destroy(gameObject, t: 0.05f);
}
```

Fonctions principales de Projectiles.cs

Maintenant munis d'une classe permettant de créer des projectiles qui peuvent exploser au contact du personnage joueur et lui infliger des dégâts, nous nous sommes lancés dans la création de la fonction AttackPlayer des ennemis.

```
private void AttackPlayer()
{
    agent.SetDestination(transform.position);

    transform.LookAt(player);

    if (!alreadyAttacked)
    {
        Rigidbody rb = Instantiate(projectile,transform.position, rotation: quaternion.identity).GetComponent<Rigidbody>();

        rb.AddForce(transform.forward * 32f, ForceMode.Impulse);

        alreadyAttacked = true;
        Invoke(nameof(ResetAttack),timeBetweenAttacks);
    }
}

1 usage  Aurelien Soula
private void ResetAttack()
{
    alreadyAttacked = false;
}
```

Fonction AttackPlayer et sa fonction auxiliaire ResetAttack

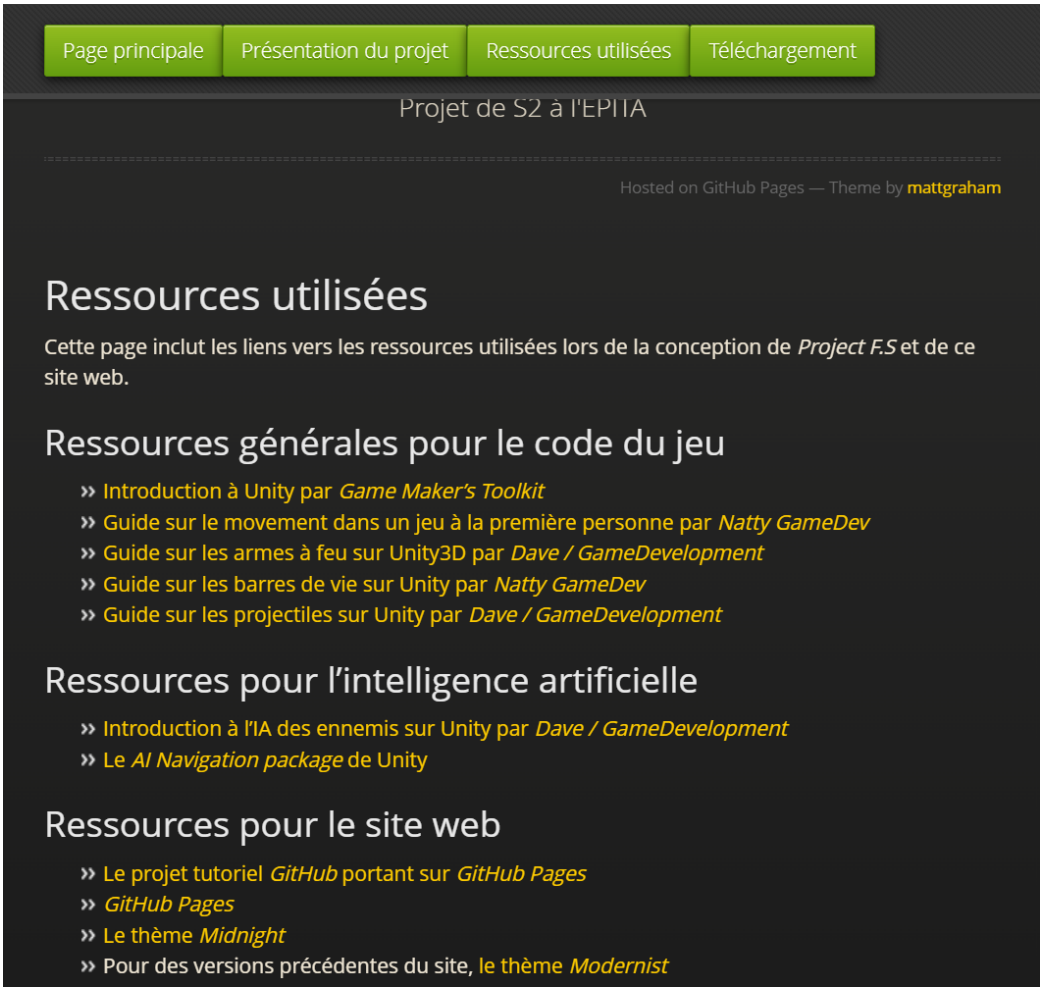
Dans cette fonction, l'ennemi s'arrête de bouger, se tourne vers le joueur, et s'il peut attaquer, instancie un projectile décrit précédemment et lui ajoute une force dans la direction du joueur avant de finalement appeler ResetAttack pour déterminer quand il pourra attaquer à nouveau. Le développement de l'IA a encore une fois été pris en charge par Aurélien.

4.7 Outils de debug

Pour simplifier les tests, nous avons également ajouté des outils de debug à *Project F.S*. Ces outils sont actuellement composés de fonctions qui permettent de soigner le joueur, de faire apparaître un ennemi et de renvoyer le protagoniste à sa position initiale. Ces outils ont été mis en place par Aurélien.

4.8 Site Web

Le développement du site web du jeu suit son cours et est toujours géré par Aurélien. La liste de ressources continue de s'étoffer et la page principale a été légèrement modifiée.



Page ressources

Page principale

Le style graphique est toujours voué à être légèrement modifié une fois celui du jeu finalisé.

5 État final

Cette section décrit l'état du projet au moment de son rendu final au mois de mai.

5.1 Level-design et décoration

Partie réalisée par Martin, Thomas et Gauthier

Gauthier a tout d'abord utilisé le package nommé *Real-time CSG* qui permet la manipulation de surfaces et de *meshes* pour créer des niveaux. C'est avec l'aide de cet outil qu'il a construit le tutoriel et les deux niveaux du jeu.

Par la suite, Thomas et Martin ont tous les deux décoré les niveaux et ont placé les ennemis ainsi que les portes qui se ferment quand le joueur entre dans une pièce.

Image du niveau 2

Il y a 3 niveaux au total dans le jeu (en comptant le tutoriel) possédant chacun un agencement et des ennemis complètement différents avec une difficulté croissante de niveau en niveau. Les couleurs sont volontairement sombres pour que les ennemis, possédant des couleurs vives, soient facilement différentiables de l'environnement (cf. état de l'art).

Autre image du niveau 2

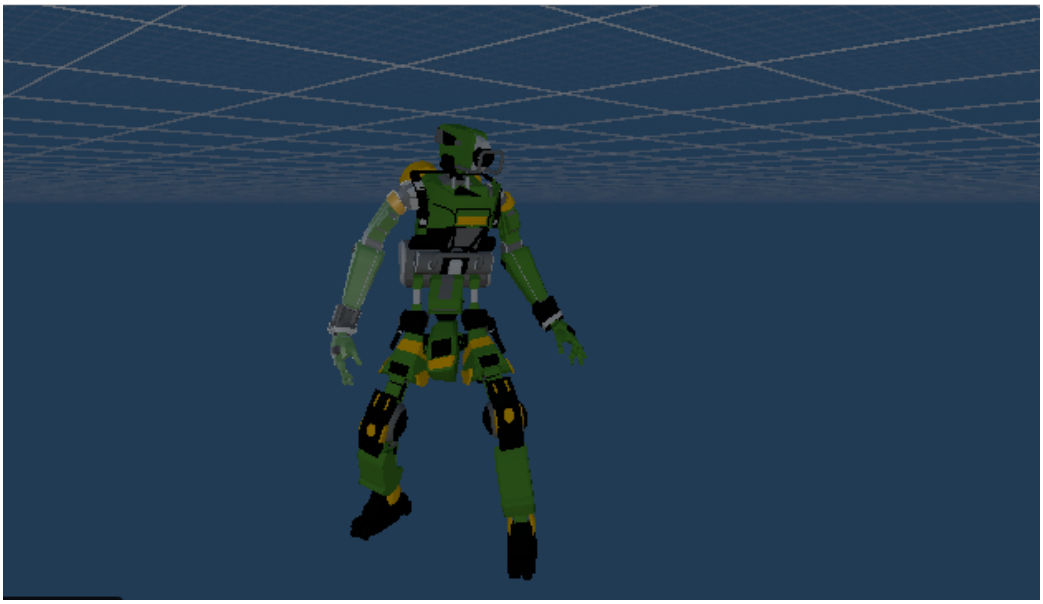
5.2 Nouveaux ennemis

En plus des ennemis à projectiles présentés précédemment, nous avons ajouté de nombreux autres types d'ennemis. En premier lieu, nous avons implémenté des ennemis infligeant des dégâts au corps à corps. Est ensuite venu l'ajout des ennemis volants, ceux-ci tirant des projectiles similaires à ceux vus plus haut. Finalement, des ennemis tirant des lasers utilisant des raycasts ont rejoint le projet. Pour pallier au problème mentionné dans la précédente section traitant des ennemis (le fait que les attaques se servant de raycasts soient quasiment impossibles à éviter), nous avons fait en sorte que ces ennemis ne commencent à tirer qu'après un certain délai et possèdent peu de points de vie, les rendant faciles à éliminer.

5.3 Modèles 3D et animations

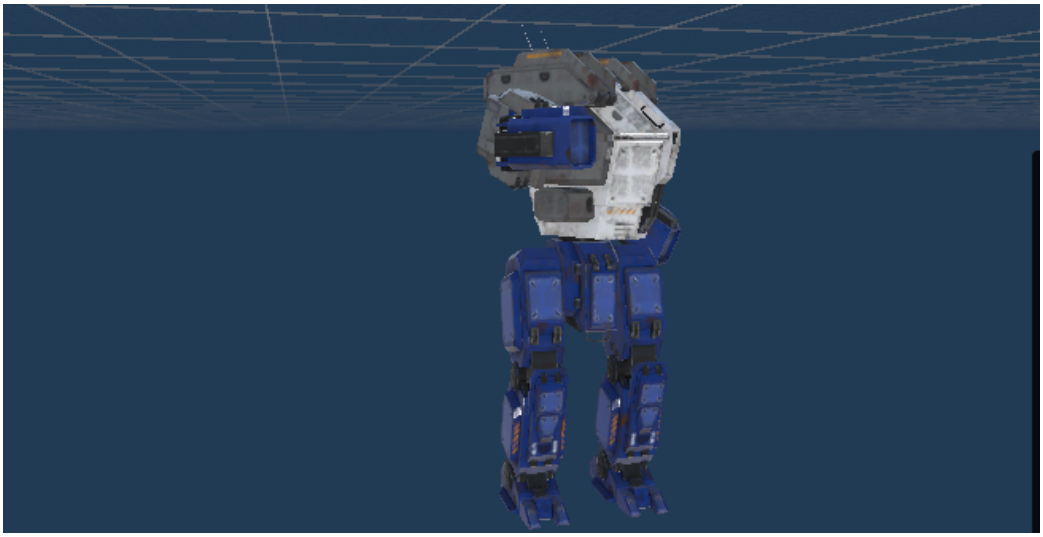
Aurélien et Gauthier ont trouvé des modèles 3D sur le *unity asset store* puis les ont animés en utilisant l'application *Mixamo*, un produit *Adobe* qui facilitent la création d'animations 3D. Ils ont ensuite ajouté ces animations aux ennemis

en utilisant le *unity animation system*. Les ennemis peuvent désormais courir, frapper au corps à corps et tirer (pour ceux qui en sont capables).



Modèle 3D de l'ennemi de base au corps à corps

Ces animations sont toutes synchronisées avec les fonctions de tirs et de dégâts pour que le joueur puisse facilement reconnaître quand un coup/une balle arrive. Les animations sélectionnées sont volontairement peu extravagantes mais suffisamment visibles pour aider au gameplay.



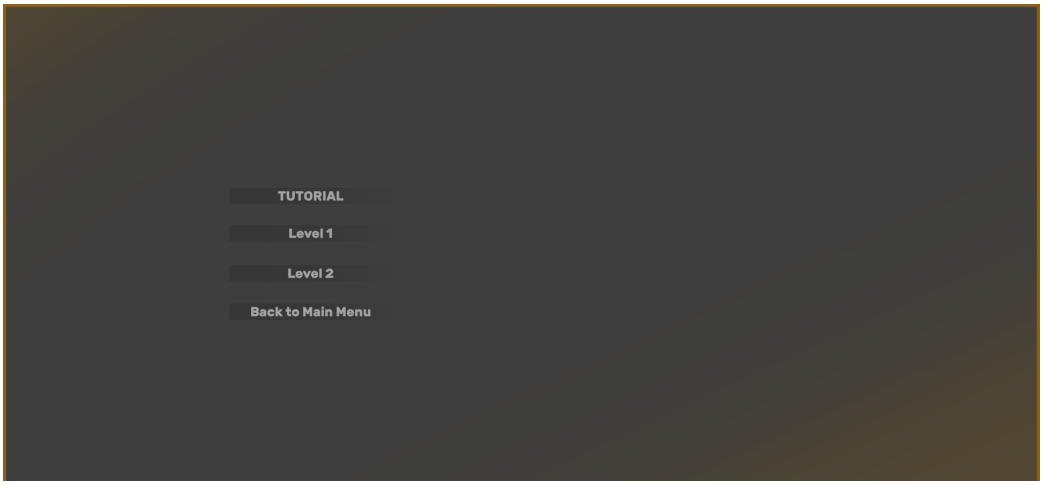
L'ennemi Tank effectuant une animation d'attaque

5.4 Finalisation des menus

Thomas s'est chargé du design et du fonctionnement des menus, des options et du menu pause en jeu. Tout d'abord, le menu principal a été la première étape :

Menu principal du jeu

Les boutons sont tous fonctionnels, permettant un accès rapide aux différentes fonctionnalités. Par exemple, si nous appuyons sur le bouton "Play", nous arrivons au menu de sélection de niveaux.



Menu de sélection de niveaux

Le menu de sélection de niveaux s'affiche grâce à des activations/désactivations de "GameObjects" Unity (ici on désactive le canva du menu principal puis on active le canva du menu de sélection de niveaux).

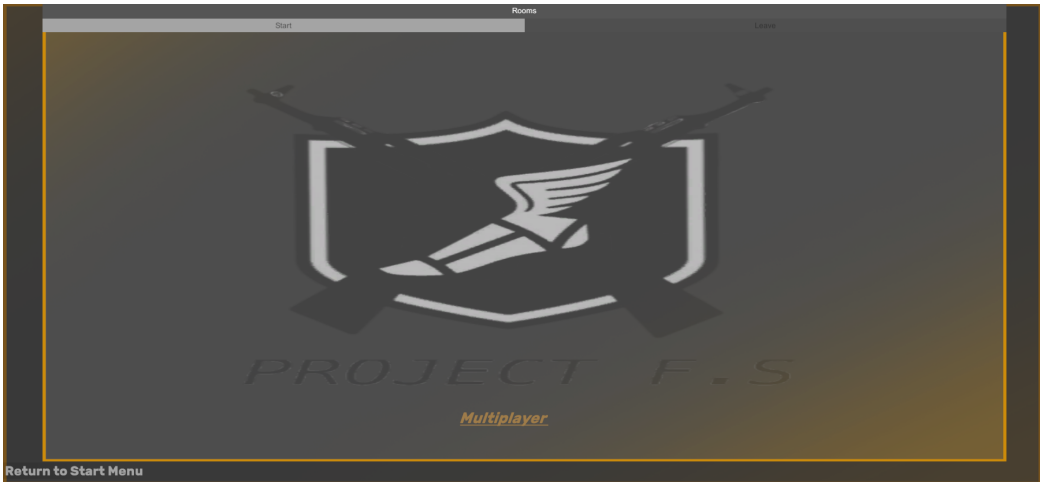
Ensuite vint la conception du menu Pause. En jeu, si nous appuyons sur "échap" :

Menu Pause

Le menu Pause vient se superposer à la caméra du joueur et arrête le temps de la partie. Dans cette phase, le joueur ne peut plus bouger mais peut interagir avec le menu pause. Il peut :

— "Resume" : quitte le menu pause et continue la partie ;
— "Option" : ouvre le menu option lui permettant de faire des réajustements ;
— "Menu" : quitte la partie et lui fait revenir au menu principal ;
— "Quit" : quitte le jeu complètement.

Enfin, le menu du mode multijoueur :

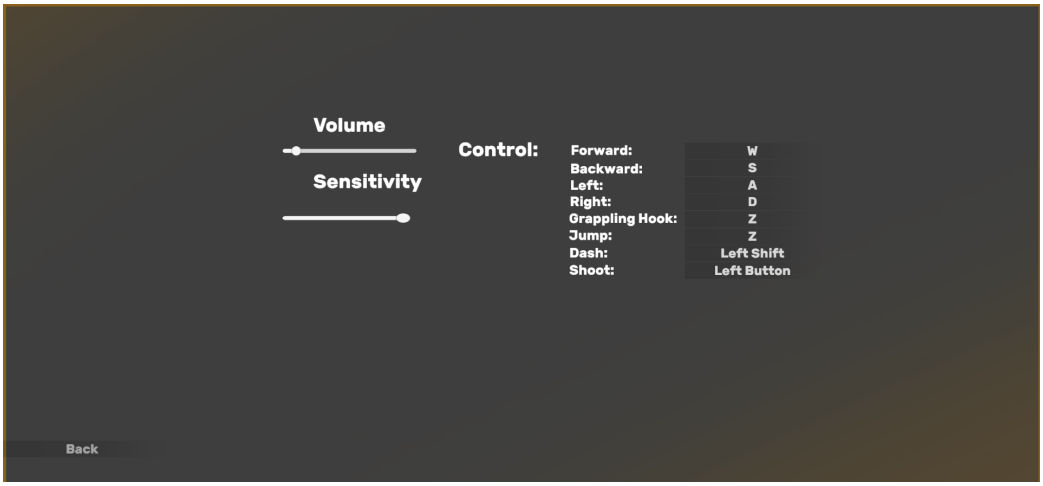


Menu Multijoueur

Étant donné qu'Alteruna vient avec son propre menu personnalisable, il nous a suffit de modifier son *prefab* pour en faire le nôtre. Dans ce menu, nous avons la possibilité de rejoindre ou de créer une "room" dans laquelle d'autres joueurs peuvent aussi rejoindre. Il nous est aussi possible de revenir au menu principal. Ce menu remplace donc le menu pause du mode solo, qui n'est évidemment pas accessible depuis le mode multijoueur.

5.5 Gestion des options

Dans la continuité des menus, Thomas s'est chargé des options du jeu :



Menu options

Dans ce menu, il est possible d'augmenter et de diminuer le volume sonore et la sensibilité de la souris grâce à des curseurs. Il est aussi possible de modifier les contrôles du personnage. En effet, lorsque vous changez une touche, par exemple "Avancer", l'*InputRebinder* écrira dans un fichier *json* la prochaine entrée de votre clavier et changera directement dans l'*InputSysteme* les touches associées.

5.6 Animations des joueurs synchronisées en ligne

Une grosse difficulté du mode en ligne a été de synchroniser les animations pour tous les joueurs. Il a donc été nécessaire de créer un nouveau script et une nouvelle classe responsable de synchroniser tous les joueurs.

Pour cela, la classe "Sub Motor" a été créée. Elle récupère localement l'état de chaque joueur (par exemple, s'il court ou saute) et ensuite synchronise cet état et l'envoie aux autres joueurs pour qu'ils puissent voir l'animation en question.

Les deux joueurs se voient donc bouger fluidement dans le mode multijoueur ce qui permet de mieux comprendre et prévoir les mouvements de l'adversaire. Cette partie a été gérée par Gauthier et Aurélien.

Animation dans le mode multijoueur

5.7 Gestion des événements et des ennemis

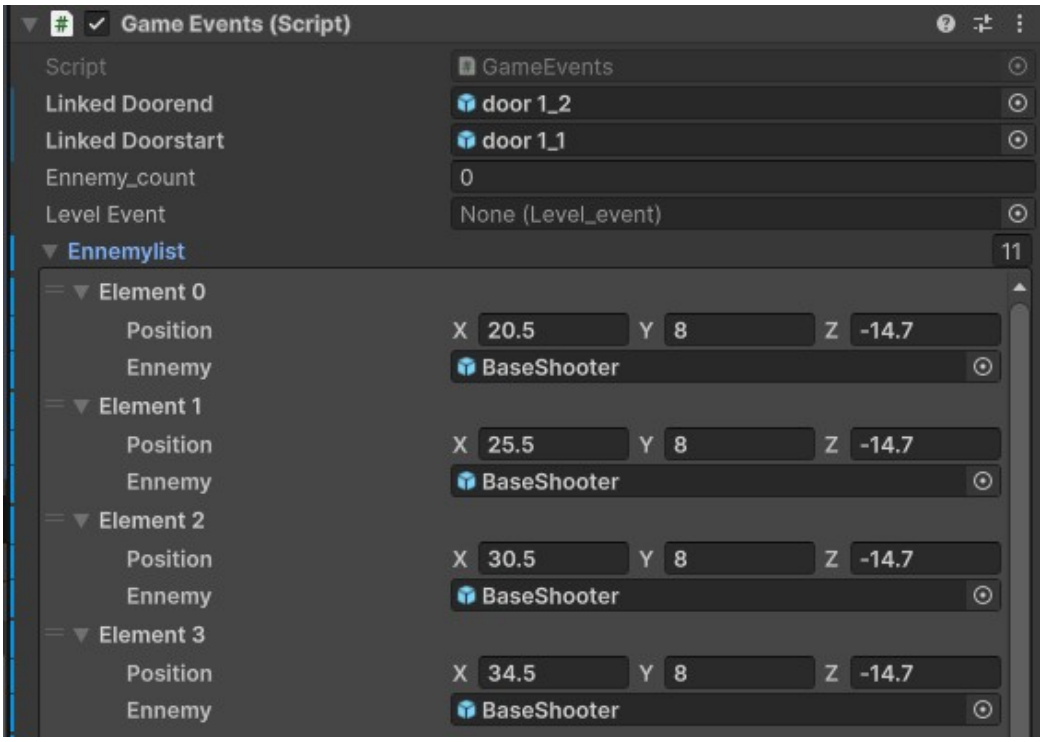
Afin d'améliorer l'interactivité du jeu, Martin a créé des événements qui ont ensuite été changés en simples fonctions par souci de paramètres. En effet, certains événements avaient besoin de paramètres, nous avons donc trouvé plus simple de passer à des fonctions.

Trois événements ont été créés : Une zone de mort où, si un joueur entre dedans, il retourne au départ du niveau ou à un checkpoint. Cet événement est utilisé dans notre jeu pour les pics afin d'éliminer le joueur si ce dernier rentre en contact avec eux. Cette mécanique rajoute du challenge et pousse le joueur à être vigilant en permanence.

Un autre événement très important est celui qui s'occupe des salles, avec l'apparition des ennemis et l'ouverture/fermeture des portes de cette salle. Le joueur sera donc obligé de rester dans cette salle et d'éliminer les ennemis avant de pouvoir poursuivre le niveau. Les ennemis apparaîtront au dernier moment

afin que le joueur ne puisse pas se préparer, et également pour créer un effet de surprise.

Et dernièrement, nous avons une fonction qui permet de se téléporter à la fin d'un niveau, soit vers le niveau suivant, soit vers le menu, en passant par une fenêtre de victoire.



Événement de gestion des ennemis et des portes

5.8 SFX et VFX

Pour ce qui est des SFX (effets sonores), nous avons utilisé deux musiques libres de droit, la première est celle du menu principal, elle était fournie dans les assets du pack pour l'UI du *unity asset store*. La seconde fut trouvé sur internet (voir site web pour référence). Enfin tous les autres SFX ont été trouvés sur le site de banque de sons "Splice" dans lequel Thomas possédait déjà un abonnement à des fins personnelles, il a logiquement réalisé ces ajouts.
Nous avons également ajouté des VFX (effets visuels) au jeu. Ceci inclut des déplacements de l'arme lorsque la caméra ou le personnage joueur se déplace,

ainsi que des tremblements de l'écran selon les tirs. Ces améliorations ont été majoritairement effectuées par Aurélien.

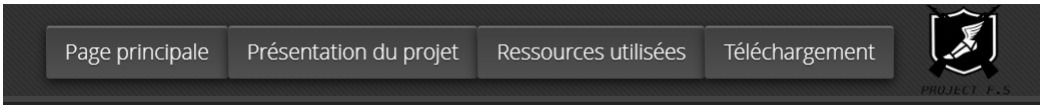
5.9 Site Web

La finalisation du site web a été gérée par Aurélien. La page principale fut entièrement remaniée. Elle a désormais une apparence similaire à l'écran titre du jeu et des animations où les différents éléments prennent leur place quand la page est chargée.

Nouvelle page d'accueil du site web

Nous avons passé du temps à faire des recherches et des essais pour que la page d'accueil s'adapte bien à une grande variété de zooms et de tailles d'écran.

Les autres pages ont également subi des modifications, la plus visible étant la couleur des boutons, qui est passée de vert à gris. Nous avons également ajouté le logo de notre jeu à la barre de navigation, celui-ci renvoyant également à la page d'accueil. La *favicon* a également été mise à jour avec notre nouveau logo.



Nouvelle barre de navigation

Toujours dans le but d'être plus en accord avec la direction artistique du jeu, la couleur des hyperliens a été changée de jaune, une couleur qui n'est que très peu présente dans *Project F.S*, à bleu.

Ressources générales pour le code du jeu

» Introduction à Unity par *Game Maker's Toolkit*
» Guide sur le movement dans un jeu à la première personne par *Natty GameDev*
» Guide sur les armes à feu sur Unity3D par *Dave / GameDevelopment*

Extrait de la page des ressources utilisées

La section "Présentation du jeu", qui incluait jusqu'ici uniquement la présentation des membres, a été étoffée avec une présentation de l'historique des réalisations et des solutions trouvées pour surmonter les problèmes.

Présentation du projet

Le jeu

Project F.S est un jeu vidéo de type Fast FPS/Movement shooter. Celui-ci sera principalement axé sur le gameplay et plus particulièrement sur les déplacements du personnage joueur dans l'environnement. L'histoire se déroule dans le cadre futuriste d'un laboratoire, plus particulièrement dans des chambres de tests – tests auxquels le joueur est forcé de participer. Vous drevez donc affronter les épreuves sur votre route pour pouvoir survivre dans cet environnement hostile.

Chronologie de réalisation

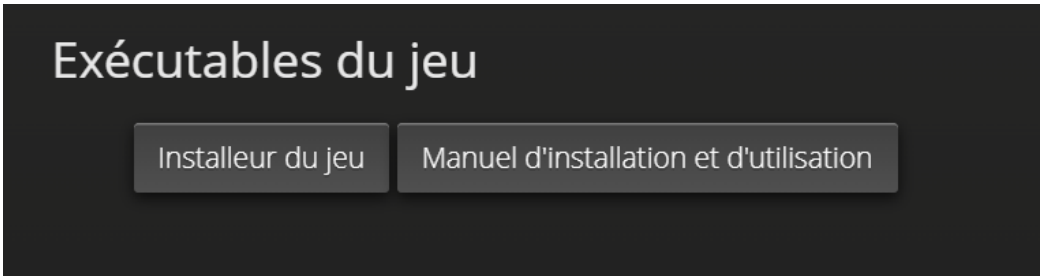
– Idée Initiale du Jeu
– Conception et implémentation du moteur physique
– Conception et implémentation de l'UI et des insterfaces
– Level-design et Importation d'assets
– Finalisation et équilibrages

Problèmes rencontrés

– Difficultés lors de l'implémentation des annimations
– Difficultés lors de l'implementation du mode multijoueur
– difficultés lors de la syncronisation des annimations du mode multijouer

Extrait de la page de présentation

L'installeur a été ajouté à la page "Téléchargement" ainsi que ce rapport.



Extrait de la page de téléchargement

6 Conclusion

Notre jeu *Project F.S* a pris forme dans le temps imparti pour ce projet. Nous sommes fiers d'avoir réussi à relever le défi de réaliser un projet abouti en quelques mois.
Nous avons réussi à atteindre les objectifs principaux du cahier des charges : nous avons créé un Fast FPS avec trois niveaux solo ainsi qu'une zone multijoueur.
Nos inspirations (Doom, Ultrakill, Portal...) sont reconnaissables, notamment dans les graphismes mais aussi dans la façon de jouer.
Notre site web est également abouti. Il contient toutes les informations relatives au jeu ainsi que les ressources qui nous ont permis de le réaliser.
Nous avons également dû nous adapter. Par exemple, nous nous sommes dirigés vers une autre implémentation du multijoueur que celle originellement prévue après s'être mieux renseignés.
Notre façon de travailler en groupe s'est aussi grandement améliorée au cours de ce projet qui nous a enseigné beaucoup de choses sur de nombreux plans.
Nous espérons que vous prendrez autant de plaisir à jouer à *Project F.S* que nous en avons pris à le réaliser.