Rapport de soutenance

PROJECT F.S

MTGA

Martin Molines

Thomas Ma-Paw-Youn

Gauthier Ruellan

Aurélien Soula

Mars 2025

Table des matières

1 Introduction **1**
1.1 Rappel des spécifications du projet 1
1.2 Rappel des tâches 2
1.3 Rappel de l'état de l'art 2

2 Avancement du projet **3**
2.1 Amélioration de la physique 3
2.2 Mécaniques de jeu 4
2.2.1 Changement d'armes 4
2.2.2 Dash 5
2.3 Map 8
2.4 Écran titre 13
2.5 Interface utilisateur 15
2.6 Amélioration des ennemis 16
2.7 Outils de debug 19
2.8 Site Web 19

3 Conclusion **21**

1 Introduction

Depuis la dernière soutenance, notre équipe a avancé sur de nombreux aspects du projet. Dans ce rapport, nous allons faire le point sur l'état actuel du jeu, de nos progrès, des problèmes rencontrés et des solutions que nous avons trouvées. Nous aborderons les mécaniques de jeu, la physique du personnage, le système d'interface utilisateur et l'intelligence artificielle pour les ennemis. Nous parlerons également de l'avancée du level design et de nos choix techniques, qui ont influencé le développement.

1.1 Rappel des spécifications 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 protagoniste enchaînera 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.

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.

1.2 Rappel des tâches

Tâches	Martin	Thomas	Gauthier	Aurélien
Scénario	R		S	
Musique et sons		R		S
Multijoueur		S	R	
IA	S			R
Graphismes/Textures		R	S	
Site Web	S			R
Mécaniques de Gameplay	R			S
Moteur Physique			R	S

1.3 Rappel de l'état de l'art

Une de nos plus grandes inspirations pour *Project F.S* est l'un des jeux les plus connus dans le domaine du FPS : *Doom*. Le premier titre de la série a été publié en décembre 1993 et la licence est aujourd'hui détenue par l'éditeur Activision. *Doom* est une référence dans le genre des FPS dits "Fast-FPS" ; en effet, il a tellement influencé l'industrie que les jeux ayant repris ce type de gameplay sont souvent qualifiés de "*Doom*-like". *Project F.S* ne sera pas épargné : notre jeu se veut rapide, fluide et jouissif, tout comme l'a été *Doom*.

L'inspiration la plus visible quand il s'agit de gameplay est le célèbre ancien concurrent de *Call of Duty* : *Titanfall*. Ce dernier, publié en 2014 par le studio *Respawn Entertainment* et édité par *Electronic Arts*, a pendant très longtemps essayé de faire de l'ombre au géant du FPS *Call of Duty*. En effet, dans *Titanfall*, c'est aussi la verticalité des combats qui est mise en avant. Le grappin est un élément essentiel dans la liberté de mouvement du joueur et dans sa capacité à s'élever dans les airs.

Enfin, une inspiration qui est cette fois visible dans la direction artistique de *Project F.S* est le très célèbre jeu de réflexion : *Portal*. Ce jeu, développé et édité par le monument du jeu vidéo *Valve Software* en 2008, raconte l'histoire d'une prisonnière, forcée par une intelligence artificielle maléfique à effectuer des tests avec un engin permettant de créer des portails "magiques" qui se relient. L'idée des "salles de tests" cubiques et le concept de "rat de laboratoire" sont repris dans l'agencement des cartes de *Project F.S* et dans l'histoire du jeu.

2 Avancement du projet

2.1 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;
}
```

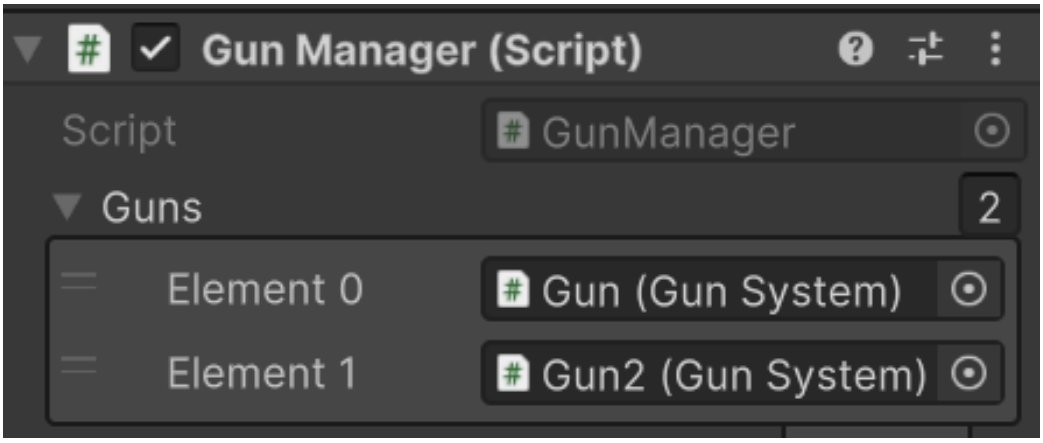
La partie de la fonction qui annule la vélocité aérienne

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.

2.2 Mécaniques de jeu

2.2.1 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.

2.2.2 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;
}
```

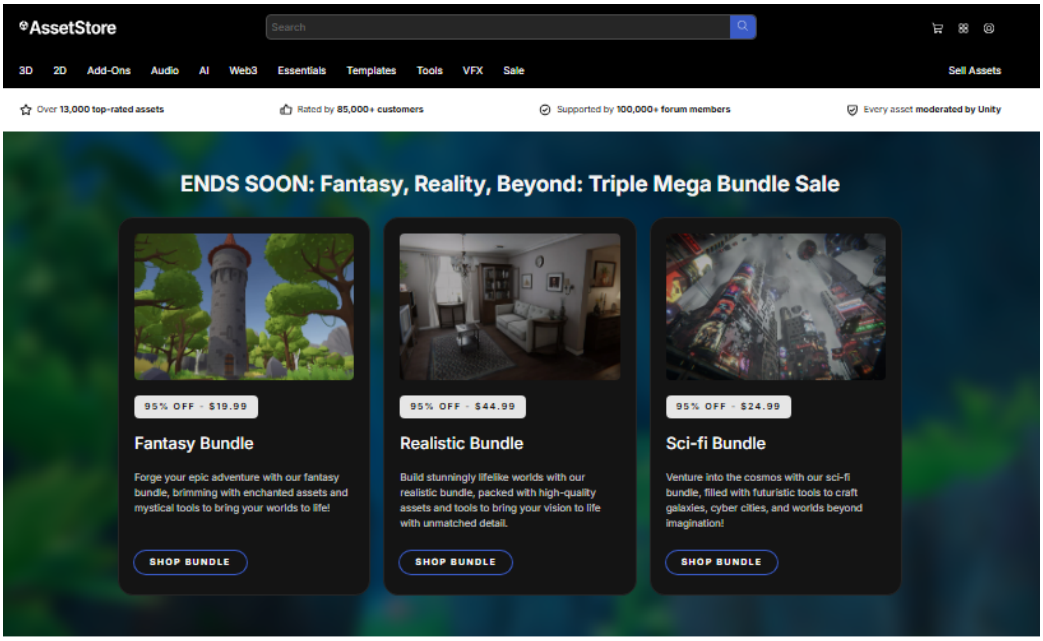
Partie de la coroutine responsable de la détection du temps et du déplacement

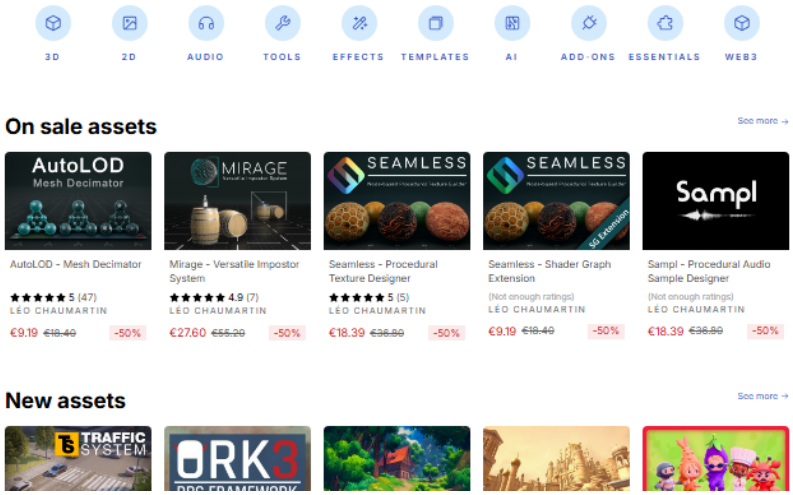
Cette partie présente l'intérêt principal de la coroutine. Elle permet d'exécuter les instructions dans l'intervalle de temps de durée "dashTime" qui commence

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

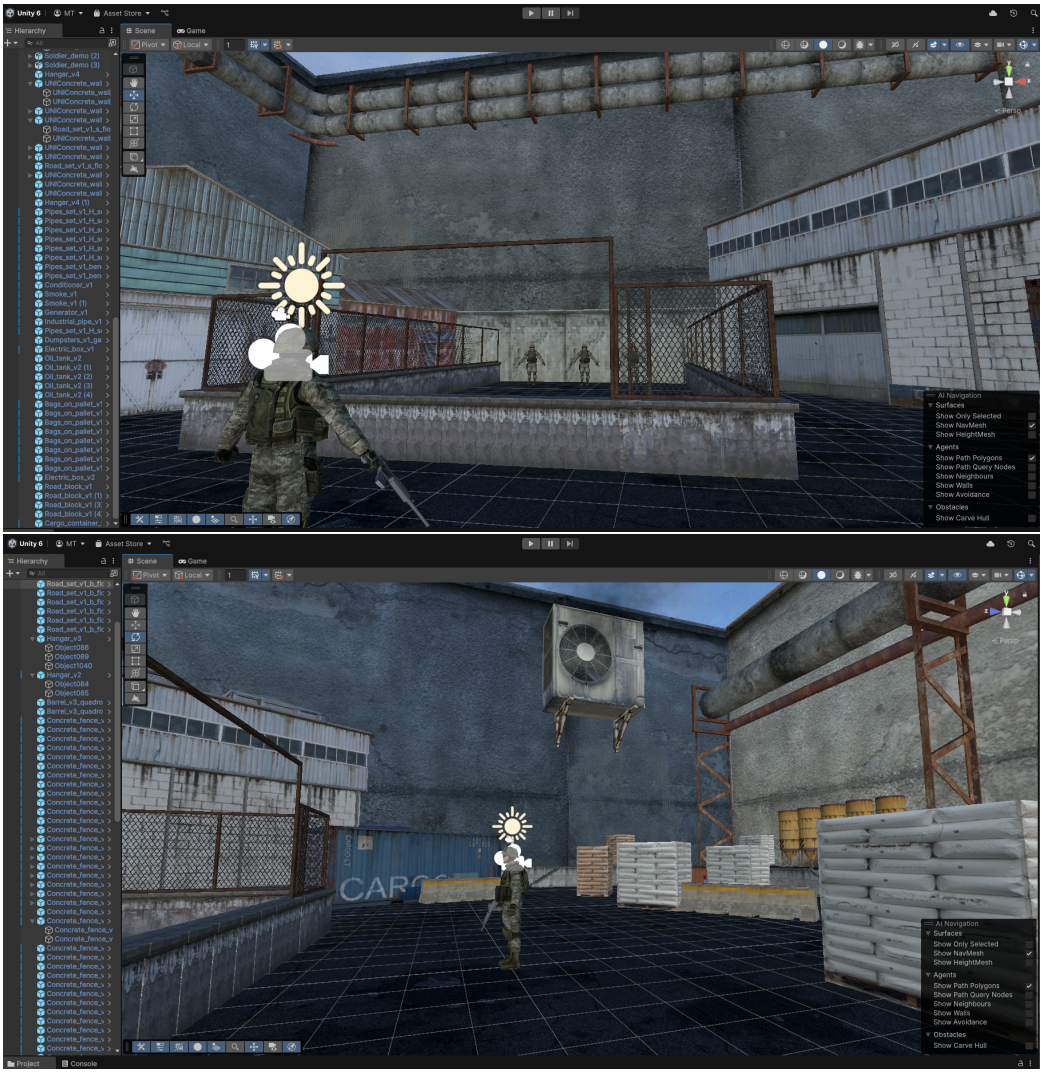
2.3 Map

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.

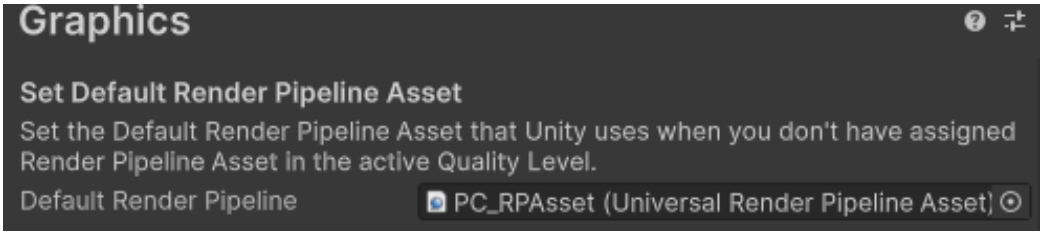




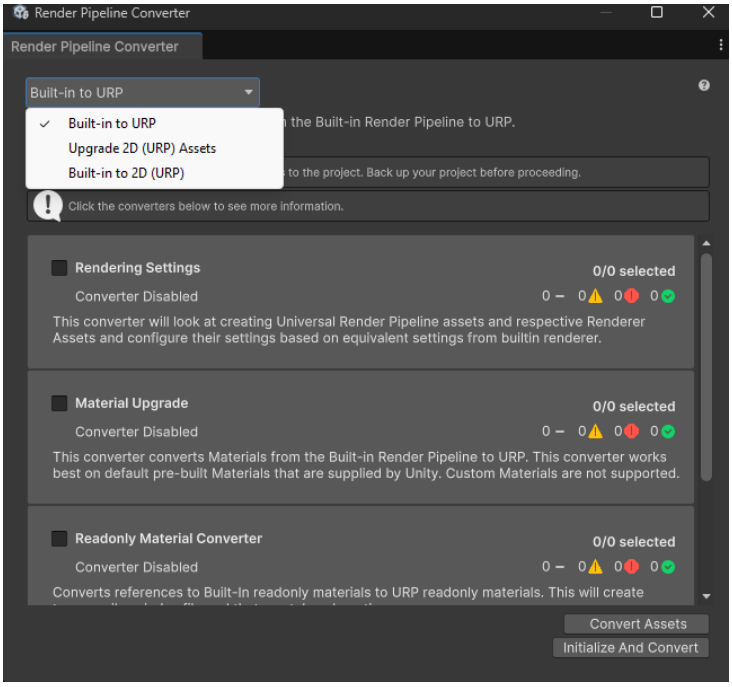
Voici un prototype de map que nous avons imaginé comme "lobby" ou "sandbox" pour que le joueur puisse tester les armes, ou tout simplement, s'amuser seul :



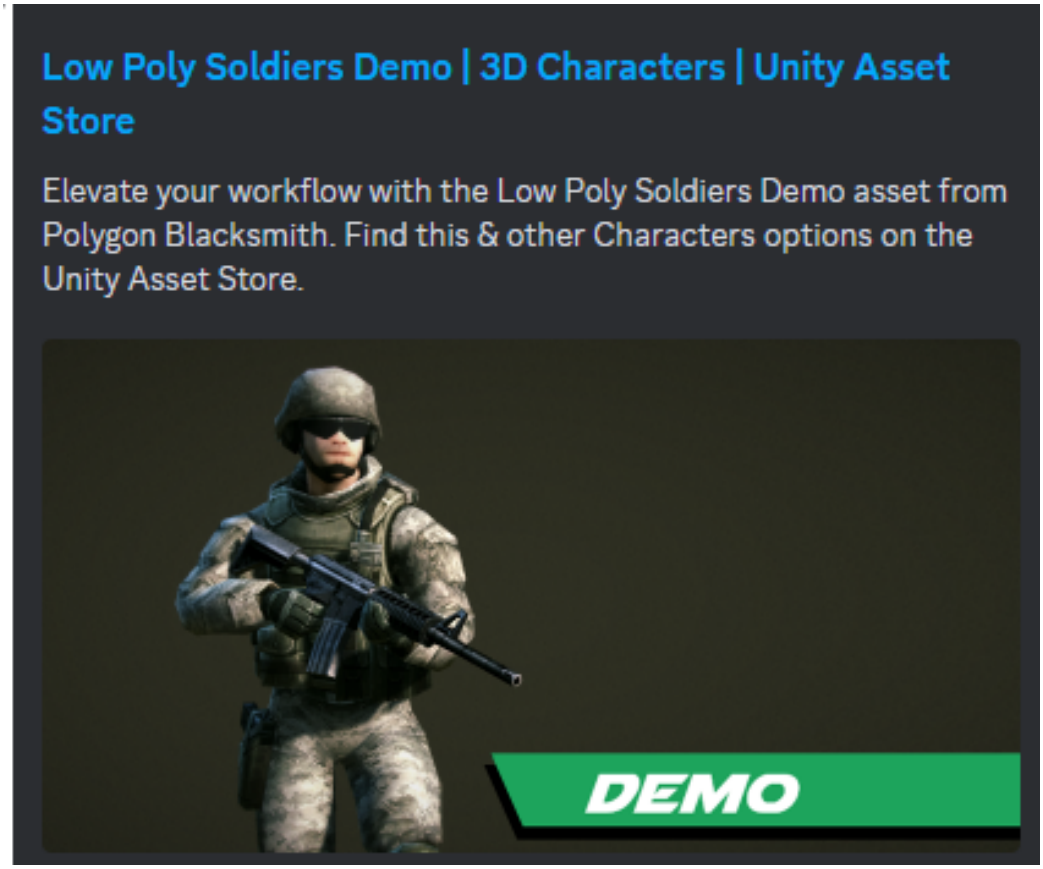
Comme vous pouvez le voir sur les screenshots, la direction artistique est bien différente de celle annoncée au départ. Cela est dû au problème esquissé précédemment : Unity dispose de plusieurs rendering pipelines, tels que URP (Universal Rendering Pipeline), Built-in Pipeline, etc. Au départ, par méconnaissance 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.

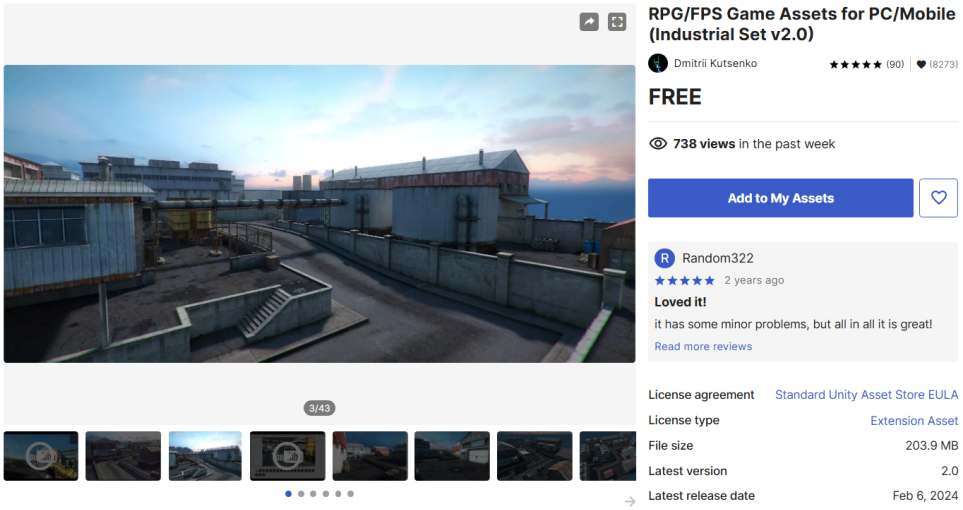


Nous avons donc décidé d'utiliser Built-in comme rendering pipeline, qui, comme le dit le nom, est la pipeline de base de Unity et est surtout utilisée pour les projets sur PC. Le problème étant qu'il existe un convertisseur automatique directement dans Unity pour convertir du Built-in vers URP, mais 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

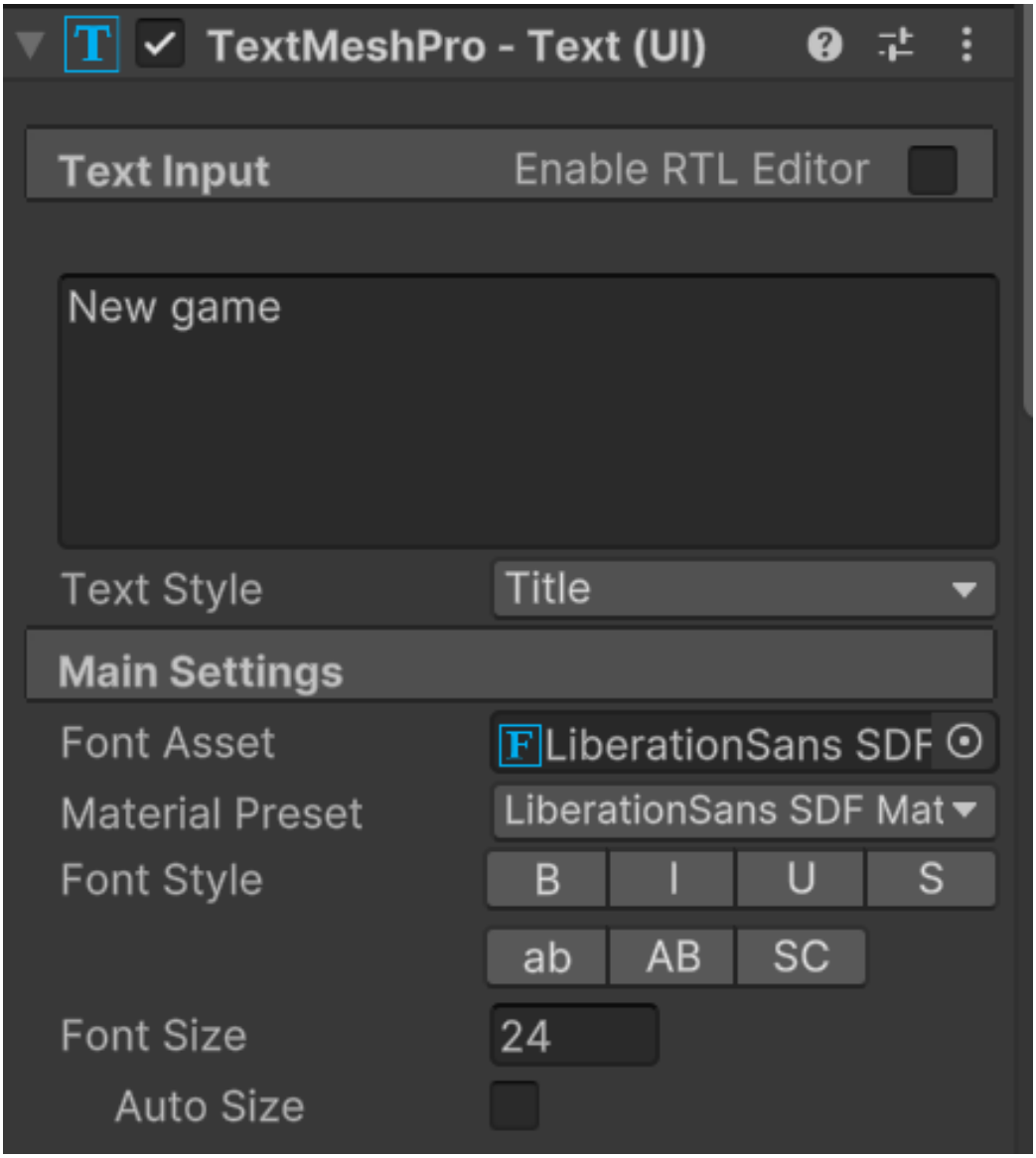
2.4 Écran titre

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

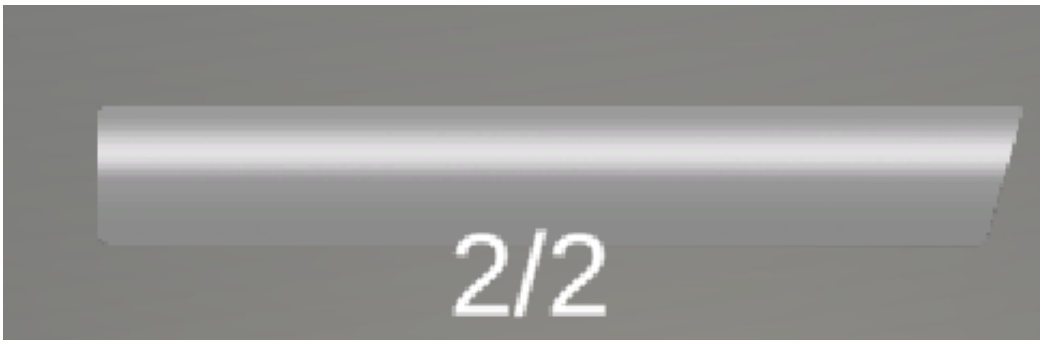


Le composant TextMeshPro

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

2.5 Interface utilisateur

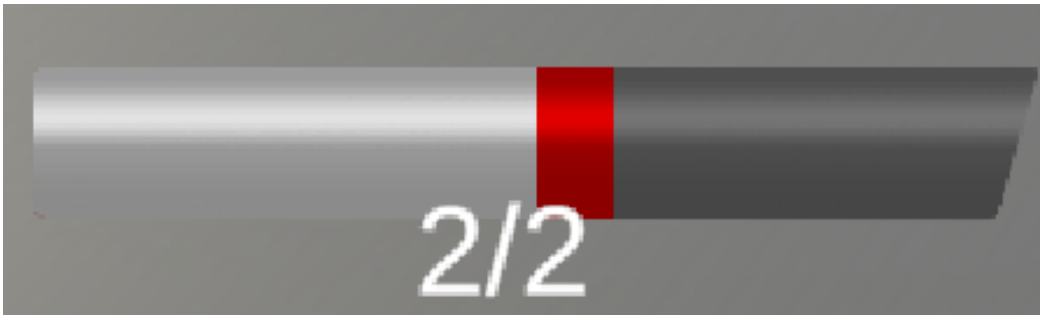
Nous avons mis en place une barre de vie et un compteur du nombre de dash disponibles, situés dans la partie inférieure de l'écran et visibles à tout moment.



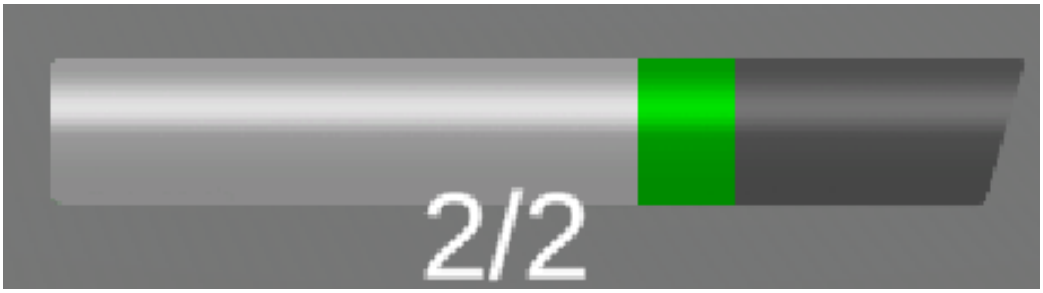
La barre de vie et le compteur de dash

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.

2.6 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

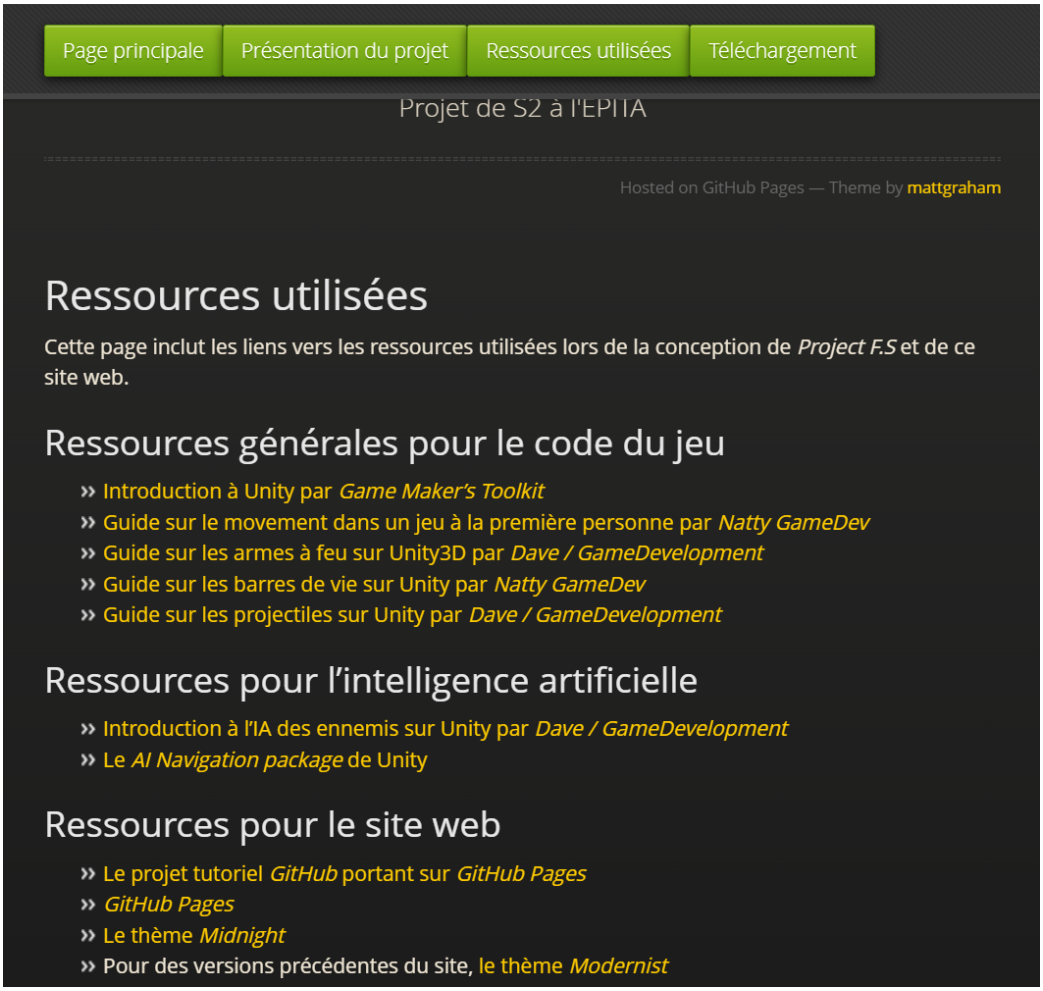
Dans cette fonction, l'ennemi s'arrête de bouger, se tourne vers le joueur, et s'il peut attaquer, instancie un projectile décrit précédemment et lui ajoute une force dans la direction du joueur avant de finalement appeler ResetAttack pour déterminer quand il pourra attaquer à nouveau. Le développement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

2.7 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.

2.8 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.

3 Conclusion

Ce rapport montre les progrès significatifs réalisés sur notre projet, en particulier en ce qui concerne le gameplay, la physique et l'interface utilisateur. Nous avons fait face à de multiples problèmes techniques et avons travaillé sur nos méthodes pour une progression efficace. Nous sommes conscients que certaines parties du code ne respectent pas encore certaines conventions et nous comptons effectuer une "harmonisation" de nos scripts dans un futur proche. Avec les améliorations et les ajouts qui restent à faire (multijoueur, campagne solo...), le projet est en bonne voie pour correspondre au cahier des charges au moment du rendu final.