

Rapport de la soutenance finale, Juin 2021



Membres du groupe:

- CARION-VIGNAUD Titouan
- HASS Léo
- MANONGO Claude-Cédric
- NDE Daniel Nolan Nathanael



Table des matières:

1/ Introduction:	Page
- Reprise du cahier des charges	4
- Plan de notre rapport	
2/ Avancement du projet, planning et prévision futures:	Page
→ Editeur 2D/Graphisme	Page
→ Gestion du Personnage et monstres	Page
→ L'interface	Page
→ Animateur	Page
→ Site Internet	Page
→ Ingénieur son et réseau	Page
3/ Problèmes rencontrés:	Page
4/ Conclusion:	Page

I. Introduction

A. L'origine du projet

L'idée de notre projet vient du premier jeu de type plateformer 2D se nomme "Space Panic". Sorti en 1980 sur borne d'arcade, il est le précurseur du genre qui par la suite va populariser le jeu vidéo.

Les principaux jeux de ce type existant sont les suivants : Super Mario Bros (la licence globalement) et Sonic the Hedgehog (là aussi la licence dans sa globalité) ainsi que le très populaire Céleste. Les deux premiers ont démocratisé à eux seuls les jeux-vidéos et le genre du jeu de plateforme. Enfin, bien que plus récent, Céleste a su révolutionner son genre et conquérir le cœur des joueurs. Ils ont tous des qualités qui les différencient d'un autre "plateformer".

Super Mario Bros est un jeu extrêmement qualitatif de par son intuitivité et son level-design, en effet il est réputé pour sa facilité de compréhension, tout en restant un challenge du début jusqu'à la toute fin du jeu, permettant ainsi de conserver l'intérêt du joueur le plus longtemps possible.

Sonic lui se caractérise par sa très bonne simulation de la vitesse, on se fait emporter par le personnage manette en main, donnant au joueur la satisfaction de pouvoir se mouvoir à pleine vitesse tout en conservant un contrôle optimal. Tout comme Mario, Sonic brille également de par son level-design. En effet, tous les niveaux sont construits de telle manière à ce que l'expérience de jeu puisse varier d'un joueur à l'autre car il est possible de finir un seul niveau de différentes manières selon les chemins empruntés. Beaucoup de structures similaires aux loopings, aussi au service de la vitesse.

Enfin, Céleste brille sur quasiment tous les aspects, les développeurs ont minutieusement conçus les mouvements du personnage principale, Madeline, de façon que chaque mouvement réalisé sur la manette soit le plus représentatif possible de la volonté et la pensée du joueur, nécessaire pour ce jeu qui se veut être un challenge : difficile à maîtriser mais facile à apprendre.

L'intuitivité est aussi une grande qualité du jeu, les contrôles sont extrêmement simples à apprendre mais nous donnent accès à une véritable liberté dans nos déplacements au sein du monde. Le monde dans sa structure fait passer un message et a donc un intérêt scénaristique tant il raconte l'ascension du personnage qui se lance à la conquête d'une montagne pleine de mystères.

B. La nature du jeu

Nous sommes tous facilement tombés d'accord sur l'idée de faire de notre projet un jeu vidéo. Notre jeu vidéo sera un plateformer : il s'agit plus précisément d'un genre de jeu où le joueur incarne un avatar qui doit se déplacer sur des plateformes suspendues tout en évitant différents obstacles pour atteindre un point donné.

Ensuite nous nous sommes inspirés du mythe d'Ariane dans la mythologie grecque et du domaine aérospatial en faisant le lien avec les fusées Ariane pour concevoir notre jeu vidéo. Il sera en 2D et sera constitué de 4 niveaux (cf. partie II).

Nous avons décidé que le personnage principal serait un personnage féminin du nom d'Ariane.

C. L'objet d'étude du projet

Ce projet va nous permettre de nous améliorer et de nous donner des compétences sur le travail d'équipe notamment sur la gestion et la répartition des tâches. Il nous permettra aussi de nous apprendre à gérer l'aspect esthétique et l'organisation d'un jeu vidéo comme le graphisme, les animations, les niveaux par exemple. Nous allons devoir mettre en place et mobiliser nos connaissances et nos compétences en algorithmique ainsi qu'en programmation, plus précisément en C#.

D. Répartition des tâches et objectifs initiaux

RESPONSABLE	SUPPLEANT	TACHES
NATHANIEL	Léo	Site internet
CLAUDE	Titouan	Editeur 2D/graphiste
TITOUAN	Léo	Animateur
TITOUAN	Nathanael	Ingé son
CLAUDE	Léo	Gestionnaire physique
TITOUAN	Nathanael	Scénariste
CLAUDE	Nathanael	Gestion Perso principal
LEO	Titouan	Gestion Perso mob
CLAUDE	Léo	Gestion Perso Boss
CLAUDE	Nathanael	Gestion Obstacle
TITOUAN	Claude	Interface
LEO	Nathanael	Réseau

	Première soutenance	Deuxième soutenance	Troisième soutenance
Site internet	50%	100%	100%
Editeur 2D/graphiste	20%	50%	100%
Animateur	20%	50%	100%
Ingé son	0%	10%	100%
Gestion perso	40%	60%	100%
Interface	40%	60%	100%
Réseau	0%	20%	100%

E. Plan du rapport de soutenance

Dans la première partie, nous montrerons les différentes étapes du projet dans chaque aspect en fonction de ce qui avait été prévu lors du cahier des charges et des deux premières soutenances.

Ensuite, nous ferons un bilan sur le déroulement du projet où nous parlerons de nos difficultés et des aspects négatifs mais aussi des aspects positifs.

Enfin, nous terminerons ce rapport par une conclusion sur ce que nous avons appris de ce projet.

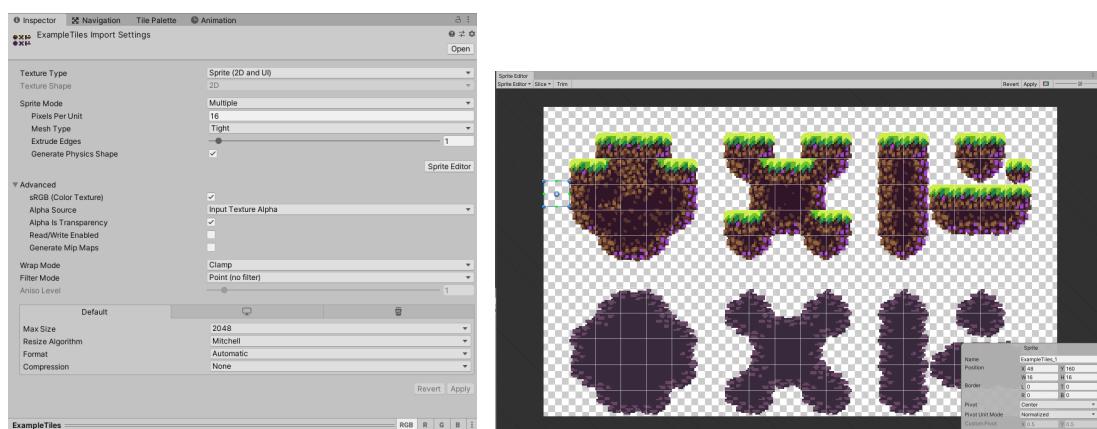
II. Avancement du projet, planning et prévision futures:

A. Editeur 2D et Graphismes

En ce qui concerne la partie Edition 2D et du graphisme, nous avons terminé les niveaux 2 et 3, grâce aux tilemaps présentent ci-dessous

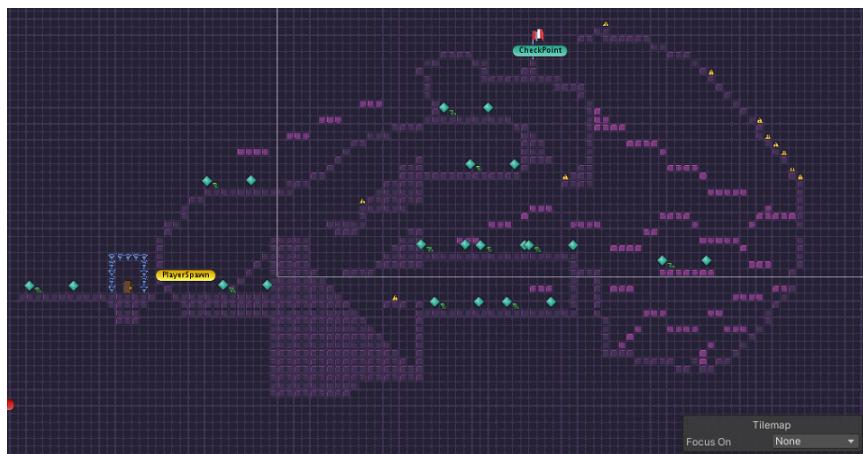
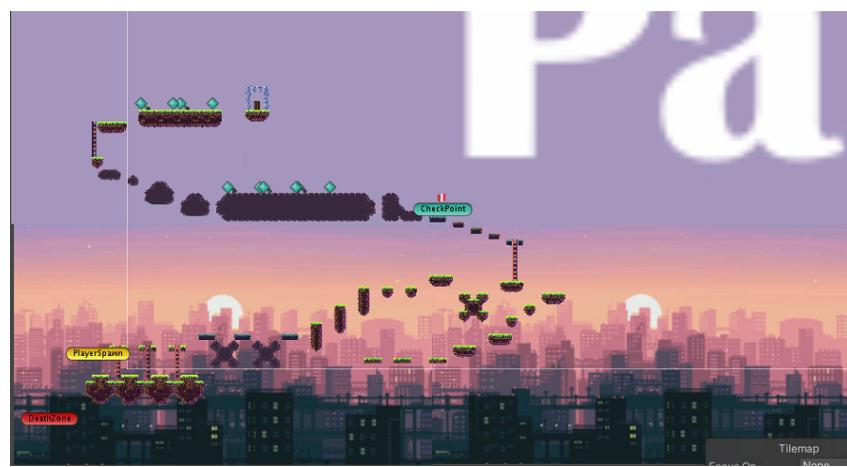
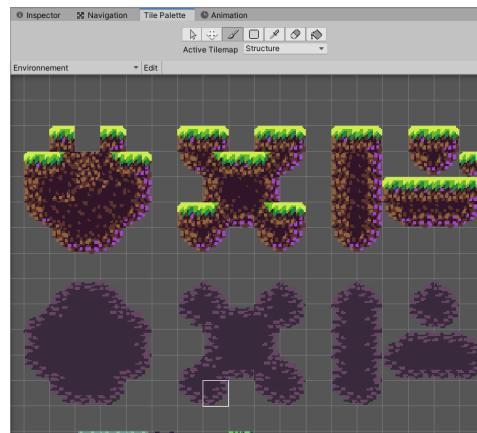


On a pu les diviser en petites cellules grâce au “Sprite Editor”.



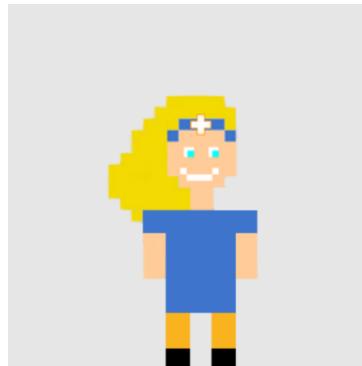
Ce qui nous a permis par la suite de les placer sur l'interface du niveau comme nous le souhaitions. Interface qui fait office de grille que nous obtenons en créant une tilemap ce qui génère automatiquement une

grille. On sélectionne alors un carreau parmi la palette qui s'est générée de par la partition de cellules et c'est avec cela que nous avons pu construire les niveaux 2 et 3.



B.Gestion du Personnage et des monstres

Pour cette troisième soutenance, nous avons commencé par remplacer le sprite de test par le sprite d'Ariane que nous avons réalisé.



(Sprite d'ariane lors de la position d'arrêt)

Nous avions, pour la soutenance précédente, créé un monstre allant d'un point A à un point B. Nous avons pour cette soutenance ajouté des monstres dans tous les niveaux de notre jeu.



(photo de quelques monstres dans le niveau 2)

Nous n'en avons pas fait plus pour ce qui est de la gestion des personnages et monstres, étant donné que nous avions la base déjà faite nous nous sommes concentrés sur les animations. (Voir section "Animateur")

C. L'Interface

Dans le jeu :

Pour permettre à l'utilisateur de savoir comment jouer nous avons implémenter des indicateurs en bas de l'écran, ces indicateurs sont composés du texte "Monter" suivi des icônes "E" et "flèche du haut"; du texte "Sauter" suivi de l'icône "espace"; du texte "Gauche" suivi de l'icône "flèche gauche"; du texte "Droit" suivi de l'icône "flèche droite" et enfin du texte "Pause" suivi de l'icône "échappe".

En haut de l'écran nous avons implémenté une barre de vie qui permet à l'utilisateur d'avoir un repère sur le niveau de vie du personnage. Cette barre de vie est composée d'une bordure fixe et d'une glissière qui change de couleur du vert au rouge lorsque le joueur subit des dégâts.

Le script permettant de faire fonctionner la barre de vie, est composé de deux parties, la première contient deux fonctions, l'une *SetMaxHealth()* pour remplir la barre à son maximum et d'initier la couleur au vert, l'autre *SetHealth()* permet de réduire la glissière grâce à un int et de déterminer la couleur de celle-ci par rapport à sa nouvelle taille.

Ces deux fonctions sont utilisées dans la deuxième partie située dans le script de la vie du personnage, lorsque le jeu commence la barre de vie est instancié à son max grâce à *Start()* qui appelle *SetMaxHealth()*, la barre est mise à jour grâce à la fonction *TakeDamage()* qui appelle *SetHealth()* avec la nouvelle vie du personnage.
En haut de l'écran nous retrouvons aussi un texte qui informe l'utilisateur du niveau dans lequel il se trouve.



Menu Pause :

Le menu *Pause* se déclenche sur la touche “Echap” du clavier peu importe la partie du jeu où l’utilisateur se trouve. Il est composé de trois boutons, “Reprendre” lié à la fonction *Resume()* qui permet de revenir à la scène où l’utilisateur était, “Menu” lié à la fonction *LoadMenu()* qui permet de revenir au menu principal et enfin “Quitter” lié à la fonction *QuitGame()* qui fait quitter l’application.



Game Over:

Le menu GameOver se déclenche automatiquement lorsque le personnage a perdu toute sa vie, à ce moment-là un écran noir apparaît avec en gros l'inscription "Game Over" puis disparait au bout de cinq secondes pour laisser place au menu principal.



Menu principal :

Le menu principal est très simple, il est composé d'un fond d'écran représentant la Terre vue de l'espace avec le nom du jeu et de deux boutons le premier "Jouer" pour lancer le premier niveau et le deuxième "Quitter" pour quitter l'application.



Cinématique :

Entre chaque niveau du jeu, nous avons implémenté des cinématiques pour faire apparaître le scénario du jeu afin que le joueur comprenne l'histoire du jeu. Ces cinématiques sont des scènes à part composées d'un lecteur de vidéo qui lance la vidéo au lancement de la scène et d'un bouton "Suivant" pour pouvoir passer à la scène suivante.



D.Animateur

1) Dégâts des ennemis

Pour les animations on a commencé par implémenter les dégâts des ennemis.

Pour ce faire, l'objectif est de détecter s'il y a eu un contact entre le collider de l'ennemi et le joueur: nous avons créé une méthode *OnCollisionEnter2D* dans la classe *EnemyPatrol* qui crée des dommages au joueur s'il est en contact avec l'ennemi.

Nous avons utilisé la classe *PlayerHealth* qui est associée à la santé du personnage en créant une variable en référence à cette classe. Ainsi nous allons utiliser la méthode *void TakeDamage* qui correspond à la quantité de dégâts que va causer l'ennemi au joueur. Donc à chaque contact avec l'ennemi, le personnage perdra une certaine quantité de vie.



```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.transform.CompareTag("Player"))
    {
        PlayerHealth playerHealth = collision.transform.GetComponent<PlayerHealth>();
        playerHealth.TakeDamage(damageOnCollision);
    }
}
```

Suite à cela nous avons mis en place un système d'invincibilité temporaire après avoir reçu des dégâts pour éviter que le joueur ne perde trop de points de vie sur une seule attaque.

Pour montrer cette courte période d'invincibilité, nous avons donc fait en sorte que le joueur clignote en jouant sur la transparence du joueur pendant un court instant après s'être fait attaquer par l'ennemi: nous

avons créé une variable *isInvincible* dans la classe *PlayerHealth* qui nous indique si le joueur est invincible ou non après s'être fait attaquer.

Nous avons donc modifié la méthode *TakeDamage* ainsi si le personnage s'est fait attaquer (*isInvincible == true*) alors on applique le système de dégâts.

```
public void TakeDamage(int damage)
{
    if (!isInvincible)
    {
        currentHealth -= damage;
        healthBar.SetHealth(currentHealth);
    }
}
```

Pour faire clignoter le personnage, nous avons utilisé une coroutine qui est une fonction qui nous permet de suspendre l'exécution et de rendre le contrôle à Unity, puis de continuer là où elle s'était arrêtée sur la trame suivante.

Nous avons créé la coroutine *InvincibilityFlash* en utilisant une variable de type *SpriteRenderer* qui contient le système de transparence. Ainsi tant que le personnage est invincible, on va modifier l'opacité du personnage en la faisant passer aux extrêmes de 0 à 1 avec un délai que nous avons défini grâce à l'instruction *yield return new WaitForSeconds* ce qui va donner l'effet de clignotement le temps que le personnage est invincible.

On met à jour la méthode *TakeDamage* en activant la coroutine.

```

public void TakeDamage(int damage)
{
    if (!isInvincible)
    {
        currentHealth -= damage;
        healthBar.SetHealth(currentHealth);
        isInvincible = true;
        StartCoroutine(InvincibilityFlash());
    }
}

public IEnumerator InvincibilityFlash()
{
    while (isInvincible)
    {
        graphics.color = new Color(1f, 1f, 1f, 0f);
        yield return new WaitForSeconds(1f);
        graphics.color = new Color(1f, 1f, 1f, 1f);
        yield return new WaitForSeconds(1f);
    }
}

```

Enfin, nous avons constaté suite à cela que le personnage était tout le temps invincible, pour régler ce problème, nous avons utilisé une autre coroutine pour que son invincibilité soit temporaire. Nous avons créé une coroutine *HandleInvincibilityDelay*: pour ce faire, nous avons mis en place un délai de 3 secondes et ensuite à la fin de ce délai, nous avons remis la variable *isInvincible* à false.

Nous avons remis à jour la méthode *TakeDamage* en ajoutant cette coroutine avec que l'effet de clignotement ne soit que temporaire le temps du délai mis dans la méthode *HandleInvincibilityDelay*.

```
public IEnumerator HandleInvincibilityDelay()
{
    yield return new WaitForSeconds(3f);
    isInvincible = false;
}
```

2) Implémentation d'échelles où l'on peut monter

Pour améliorer le jeu, nous avons incorporé des échelles dans nos différents niveaux pour permettre au personnage d'atteindre des blocs en hauteur lorsqu'ils ne sont pas accessibles par des sauts.

Pour cela nous avons rajouté les échelles en fond dans la partie “Scène” pour permettre au joueur de traverser s'il ne veut pas monter. Ensuite nous avons défini une variable booléen *isInRange* qui renvoie vrai si le joueur est à proximité de l'échelle. Pour savoir si nous avons rajouté une boite de collision plus précisément un *BoxCollider 2D* sur la hauteur de l'échelle, de plus on a activé le paramètre *isTrigger* qui s'active si le joueur a traversé cette boite de collision.

Nous avons créé un script pour les échelles nommée *Ladder* qui va contenir la classe *Ladder* qui va nous permettre de faire la partie plus technique.

Dans cette classe, on a utilisé les deux méthodes *OnTriggerEnter2D* et *OnTriggerExit2D* qui ont tous deux comme paramètre une variable de type *Collider2D*: si le joueur traverse la boite de collision alors on considère qu'il est suffisamment proche de l'échelle pour monter ainsi dans la première méthode, le bool *isInRange* devient *true* et dans la deuxième elle devient *false*.

```
18
19     private void OnTriggerEnter2D(Collider2D collision)
20     {
21     }
22
23     private void OnTriggerExit2D(Collider2D collision)
24     {
25     }
26
27     }
28 }
```

Nous avons ajouté dans la classe *MovePlayer* un indicateur d'état *isClimbing* qui est un booléen qui vérifie si le joueur est en train de monter.

Suite à cela nous avons modifié la méthode *MovePlayer* qui s'occupait initialement du mouvement du joueur à l'horizontale.

```
43
44     void MovePlayer(float _horizontalMovement)
45     {
46         if (!isClimbing)
47         {
48             Vector3 targetVelocity = new Vector2(_horizontalMovement, rb.velocity.y);
49             rb.velocity = Vector3.SmoothDamp(rb.velocity, targetVelocity, ref velocity, .05f);
50
51         if (isJumping)
52         {
53             rb.AddForce(new Vector2(0f, jumpForce));
54             isJumping = false;
55         }
56     }
57 }
```

Pour intégrer le mouvement verticale du joueur lorsqu'il est en train de monter, nous avons créé une variable de float *verticalMouvement* qui représente la vitesse de déplacement du joueur à la verticale: par la suite nous l'avons légèrement diminué par rapport à la vitesse de déplacement horizontale pour que le mouvement soit plus réaliste.

Ensuite nous avons modifié la méthode *MovePlayer* en ajoutant une condition: si le joueur est en train de monter alors le joueur ne peut ni se déplacer à l'horizontale ni sauter.

```

// référence
void Movejoueur(float _horizontalMouvement, float _verticalMouvement)
{
    if (!isClimbing)
    {
        Vector3 targetVelocity = new Vector2(_horizontalMouvement, rb.velocity.y); //calcul de la vitesse du perso
        rb.velocity = Vector3.SmoothDamp(rb.velocity, targetVelocity, ref velocity, .05f);

        if (isJumping == true)
        {
            rb.AddForce(new Vector2(0f, jumpForce));
            isJumping = false;
        }
    }
    else
    {
        //déplacement à la verticale
        Vector3 targetVelocity = new Vector2(0,_verticalMouvement); //calcul de la vitesse du perso
        rb.velocity = Vector3.SmoothDamp(rb.velocity, targetVelocity, ref velocity, .05f);
    }
}

```

Après avoir modifié la classe *MovePlayer* nous sommes retournés sur la classe *Ladder*, nous avons créé une référence pour le mouvement du joueur donc une variable de type *MovePlayer*. Nous avons remplacé la méthode *void Start* par *void Awake* parce que nous avions initialisé un comportement dans le programme. Dans ce void *Awake*, nous avons ajouté cette commande ci-dessus afin de récupérer le script du *MovePlayer*:

```

// Message Unity | 0 références
void Awake()
{
    movePlayer = GameObject.FindGameObjectWithTag("Player").GetComponent<MovePlayer>();
}

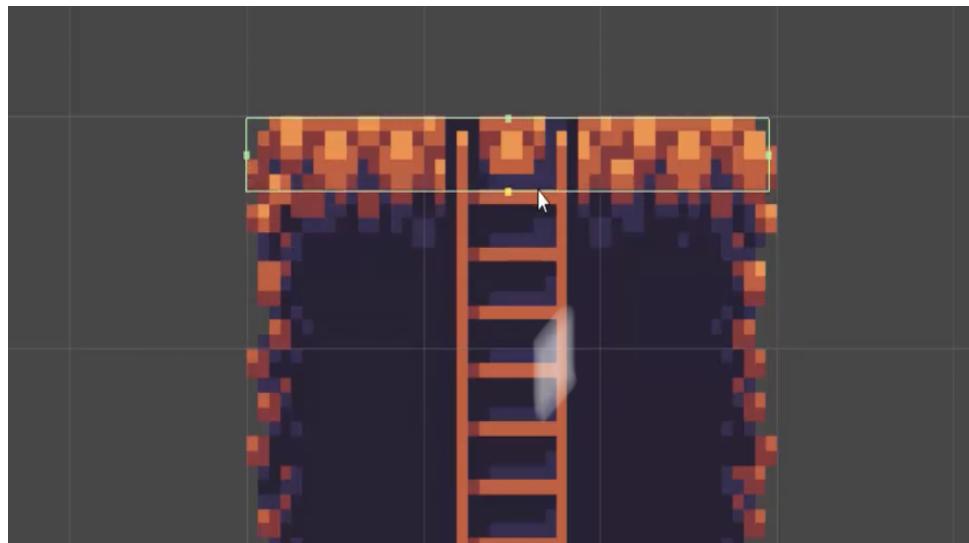
```

Ensuite nous avons utilisé la méthode *void Update* pour amorcer la montée du joueur . Si le joueur est proche de l'échelle (donc si *isInRange == true*) et s'il appuie sur le bouton E que nous avons choisi au préalable comme bouton pour la montée du joueur alors le joueur monte sur l'échelle (*movePlayer.isClimbing =true*) et il ne peut plus se déplacer à l'horizontale. Pour empêcher le joueur de continuer à monter même s'il n'est plus sur l'échelle, on fait passer le *movePlayer.isClimbing* à false dans le *OnTriggerExit2D* pour qu'il arrête de monter lorsqu'il n'est plus sur l'échelle.

En plus nous avons ajouté une autre condition (le premier if) pour permettre au joueur de descendre de l'échelle s'il ne veut plus monter finalement en appuyant sur le bouton E.

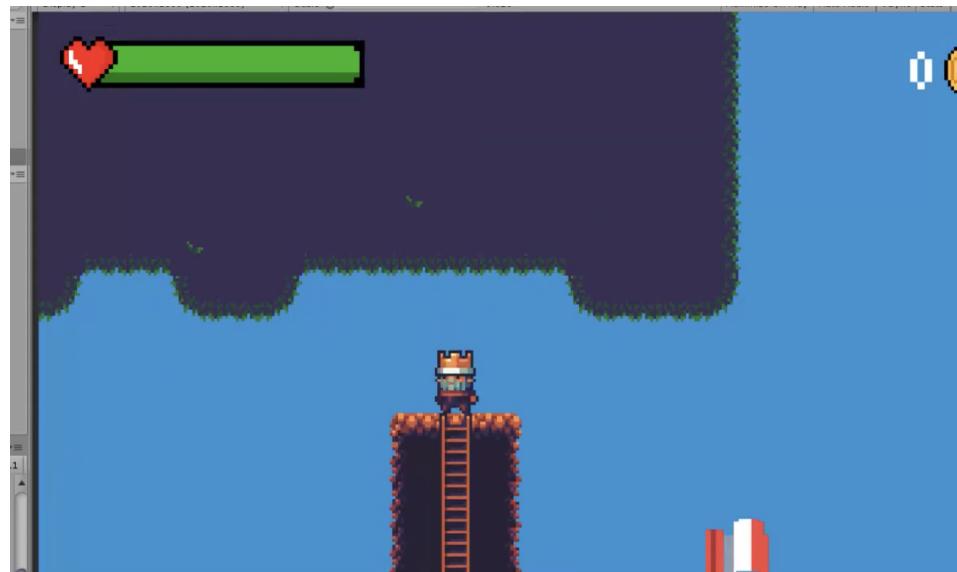
```
© Message Unity | 0 références
void Update()
{
    if(isInRange && movePlayer.isClimbing && Input.GetKeyDown(KeyCode.E))
    {
        movePlayer.isClimbing = false;
        topcollider.isTrigger = false;
        return;
    }
    if(isInRange && Input.GetKeyDown(KeyCode.E))
    {
        movePlayer.isClimbing = true;
        topcollider.isTrigger = true;
    }
}
```

Nous avons ajouté un autre collider à l'intérieur du *Ladder* sur le haut de l'échelle afin que le joueur puisse accéder au bloc qu'il souhaite atteindre parce que sans cela le joueur ne pourra pas sortir de l'échelle.



Néanmoins on s'aperçoit que si le joueur est sur l'échelle, le collider l'empêche de monter à un certain niveau ainsi l'idée est d'activer la fonction `isTrigger` pour permettre au joueur de traverser la boîte de collision lorsqu'il est sur l'échelle et de la désactiver lorsqu'il sort de l'échelle pour pouvoir se tenir dessus.

Pour ce faire, on crée une référence de type `BoxCollider2D` au collider qu'on vient de créer. Ainsi dans la méthode `Update`, on active le `isTrigger` pour la boîte de collision qu'on a rajouté (`topcollider.isTrigger = true`) et on le désactive dans `OnTriggerExit2D` (`topcollider.isTrigger = false`).



3) Zones d'élimination

Pour compléter notre jeu, nous avons créé des zones d'élimination dans nos différents niveaux. Nous nous sommes rendus que sans elles, lorsque le personnage tombait dans le vide, il tombait à l'infini et empêchait le joueur de continuer à jouer.

Pour cela nous avons défini un endroit où le personnage revenait lorsqu'il tombait dans une DeathZone en français une zone de mort. Nous avons utilisé un objet *PlayerSpawn* qui définit la zone d'apparition du personnage



Pour introduire ces zones d'élimination, nous avons créé une boîte de collision que nous avons placée dans les endroits considérés comme des zones de vide. Nous avons créé la classe *DeathZone* où nous avons utilisé la méthode *OnTriggerEnter2D*: ici si le joueur rentre en contact avec la boîte de collision alors le joueur devra revenir à la position indiqué par le *PlayerSpawn*.

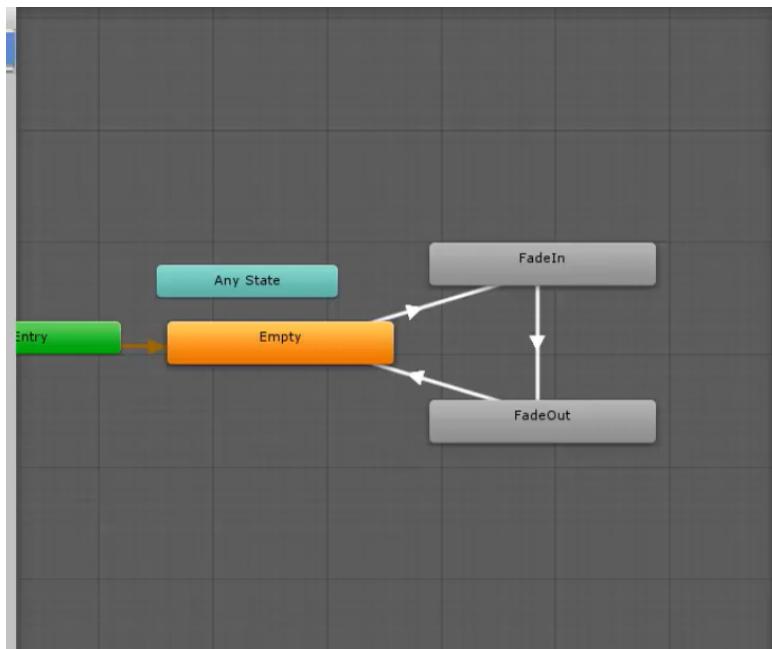
```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag("Player"))
    {
        collision.transform.position = GameObject.FindGameObjectWithTag("PlayerSpawn").transform.position;
    }
}
```

Dans le but d'améliorer ce système, nous avons créé une variable *playerSpawn* de type Transform que nous avons initialisé dans la méthode void Awake avec cette ligne de commande:

En faisant cela, nous réduisons le coût de l'opération de ce système qui était important sans cette modification.

Nous avons décidé d'améliorer les animations de ce système pour le rendre plus intéressant, nous avons ajouté une animation de fondu le temps entre la chute du personnage dans le vide et sa réapparition dans le jeu: pour ce faire, nous avons créé une variable de type Animator nommée *fadeSystem* que nous avons initialisé dans la méthode void Awake à l'aide de cette commande:

```
private void Awake()
{
    playerSpawn = GameObject.FindGameObjectWithTag("PlayerSpawn").transform;
    fadeSystem = GameObject.FindGameObjectWithTag("FadeSystem").GetComponent<Animator>();
}
```



Pour mettre en place cette animation nous avons utilisé une coroutine *ReplacePlayer* dans lequel nous avons activé l'animation de fondu et demandé au joueur de retrouver sa position initiale; nous avons introduit un court délai d'une seconde afin de pouvoir constater cette transition.

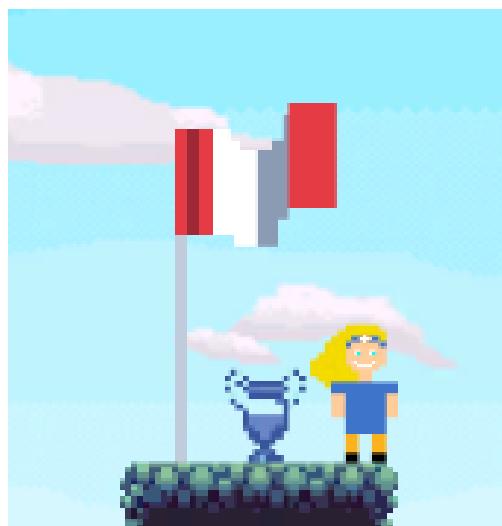
```
private IEnumerator ReplacePlayer(Collider2D collision)
{
    fadeSystem.SetTrigger("FadeIn");
    yield return new WaitForSeconds(1f);
    collision.transform.position = playerSpawn.position;
}
```

Enfin on lance la coroutine dans la méthode *OnTriggerEnter2D* si le joueur est entré en contact avec la boîte de collision qui représente la zone d'élimination.

```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag("Player"))
    {
        StartCoroutine(ReplacePlayer(collision));
    }
}
```

4) Check Points

Nous avons ensuite continué par implémenter des checkpoints dans chaque niveau. Pour cela nous avons commencé par chercher les sprites d'un drapeau pour permettre au joueur de savoir où se trouve le checkpoint. Nous lui avons fait une animation de drapeau balayé par le vent à l'aide de l'outil Animator pour avoir tout de même un côté esthétique.

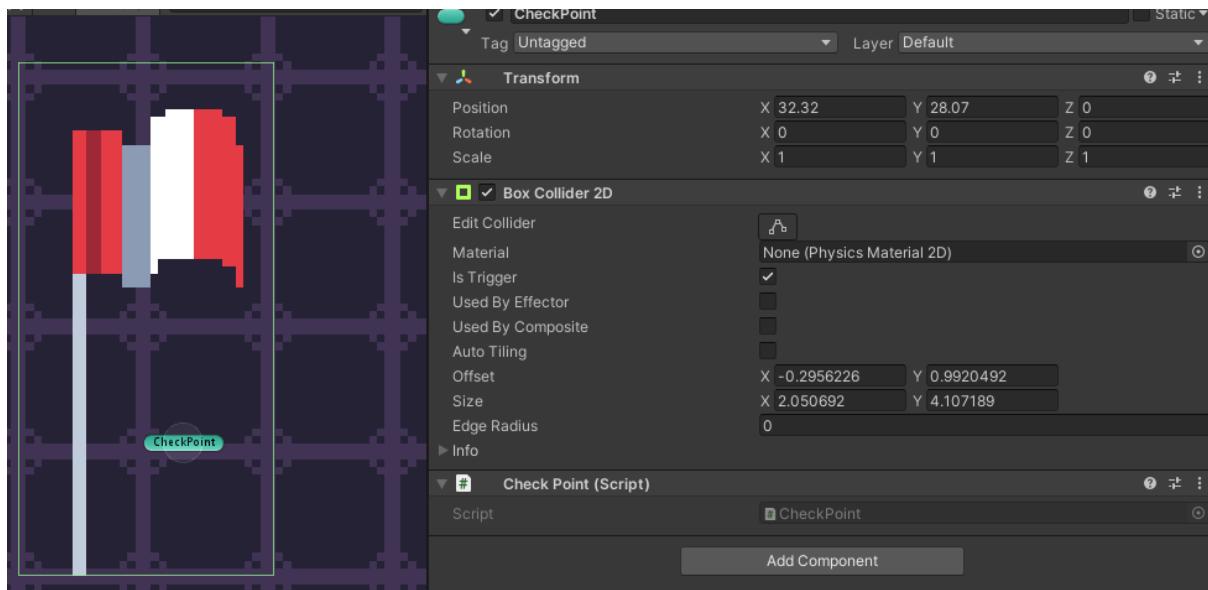


(Drapeau pour les checkpoints)

Nous avons créé précédemment un objet *PlayerSpawn* qui comme son nom l'indique permet au personnage s'il tombe dans le vide, de revenir à l'endroit où se trouve cet objet. Pour les checkpoints, nous allons tout simplement déplacer cet objet *PlayerSpawn* à l'endroit du checkpoint une fois que le joueur aura passé celui-ci.

On a d'abord commencé par créer un objet qu'on appelle *CheckPoint* pour mieux se repérer. Nous lui avons ajouté un box collider 2D pour

permettre de détecter quand le joueur passera le checkpoint. Nous lui avons également ajouté un script.



Nous avons continué par l'édition du script du checkpoint, nous avons d'abord accéder au position du *PlayerSpawn* et nous avons à l'aide de la méthode *CompareTag* détecter si le joueur rentre dans la boîte de collision. Si celui-ci rentre on met le *PlayerSpawn* à la position du checkpoint.

```
using UnityEngine;

public class CheckPoint : MonoBehaviour
{
    private Transform playerSpawn;
    private void Awake()
    {
        playerSpawn = GameObject.FindGameObjectWithTag("PlayerSpawn").transform; //pour accéder au position du playerspawn
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if(collision.CompareTag("Player")) //si le joueur entre la boite de collision
        {
            playerSpawn.position = transform.position; //on met le playerspawn a la position du checkpoint
            gameObject.GetComponent<BoxCollider2D>().enabled = false; //pour désactiver le checkpoint précédent si l'on a atteind un check point plus loin
        }
    }
}
```

La ligne `gameObject.GetComponent<BoxCollider2D>().enabled = false;` permet de désactiver la boîte de collision du checkpoint précédent pour permettre au joueur de spawn au checkpoint le plus avancé dans le niveau.

(En désactivant la boîte de collision on désactive tout simplement le checkpoint)

(le `.enabled = false` permet de désactiver le *BoxCollider2D*)

Nous avons à présent un système fonctionnel de checkPoint qui marche dans tous les niveaux. Pour ajouter ce système de checkpoint dans les autres niveaux, nous l' avons simplement dupliqué et placé à l'endroit souhaité.

5) Mort du Personnage

Concernant la mort du personnage nous avons premièrement choisi une animation de fumée quand celui-ci meurt. Nous avons trouvé une tilesheet qui nous convenait bien donc nous l'avons importée sur Unity.

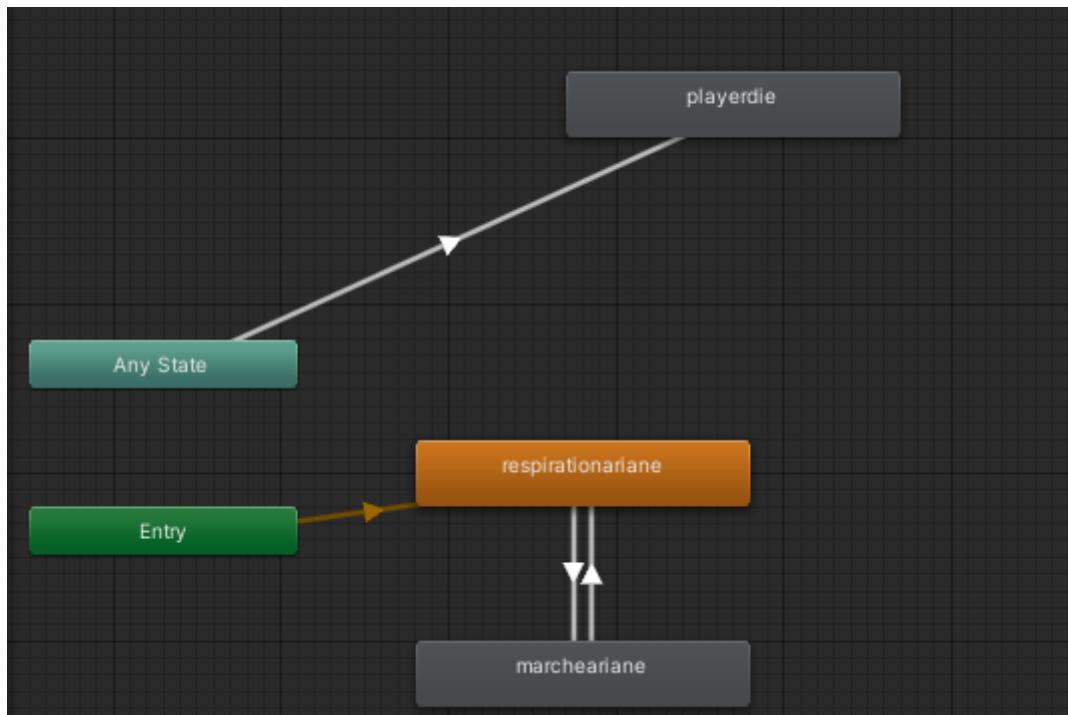


(Photo des 8 frames pour faire l'animation)

Pour réaliser l'animation, nous avons simplement superposé les images ci-dessus à la suite avec quelques secondes d'écart pour obtenir l'animation voulu.

Une fois l'animation réalisée, nous l'avons implémenté dans l'outil Animator de Unity qui nous permet de choisir quand déclencher l'animation.

Etant donné que le joueur peut mourir à n'importe quel moment, nous avons relié cette animation à *Any State* ce qui, comme son nom l'indique permet à l'animation de mort de se déclencher dans n'importe quel état du joueur.



(Photo des animations de notre personnage dans l'image *playerdie* correspond à l'animation de mort)

Nous avons continué par le script de la mort du personnage. Nous n'avons pas ajouté de nouveaux script pour cette fonctionnalité, nous l'avons simplement implémenté dans le *PlayerHealth.cs* .

Dans la méthode *TakeDamage()* ci-dessous nous avons fait vérifier si la vie actuelle du joueur est inférieure ou égale à 0 pour justement savoir si le joueur doit mourir. S'il doit mourir on appelle une méthode *Die()* qui vous sera expliquée un peu plus loin.

```
2 références
public void TakeDamage(int damage)
{
    if (!isInvincible)
    {
        currentHealth -= damage;
        healthbar.SetHealth(currentHealth);

        //vérifier si le joueur est toujours vivant
        if (currentHealth <= 0)
        {
            Die();
            return;
        }
    }
}
```

Pour la méthode *Die()* celle-ci permet de bloquer les mouvements du personnage, de jouer l'animation d'élimination vue précédemment et elle permet enfin d'empêcher les interactions physiques du personnage avec les autres éléments de la scène pour, par exemple éviter qu'un monstre pousse le joueur alors que celui-ci est mort.

```

1 référence
public void Die()
{
    Debug.Log("Le joueur est éliminé");
    // bloquer les mouvements du personnage
    MovePlayer.instance.enabled = false;

    // jouer l'animation d'élimination
    MovePlayer.instance.animator.SetTrigger("Die");

    // empêcher les interactions physique avec les autres éléments de la scène
    MovePlayer.instance.rb.bodyType = RigidbodyType2D.Kinematic;
    MovePlayer.instance.playerCollider.enabled = false;
}

```

Pour bloquer les mouvements du personnage, on fait une référence au *MovePlayer* (script qui gère les mouvements du personnage) et on le désactive à l'aide du *.instance.enabled = false*.

Pour l'animation d'élimination, on fait encore une référence au *MovePlayer* car c'est lui qui "contient" les animations du joueur, qui contient la référence à l'animator. Le *SetTrigger("Die")* va donc permettre de déclencher l'animation de mort.

Pour empêcher les interactions physique avec le monde, on fait une fois de plus référence au *MovePlayer*, on va passer le *rb* en *Kinematic* (le *rb* est le *RigidBody* du personnage c'est-à-dire sa physique et le *Kinematic* permet tout simplement d'éviter les contacts avec le monde extérieur).

Une fois cela fait il nous reste simplement à désactiver la boîte de collision du personnage d'où la ligne:

MovePlayer.instance.playerCollider.enabled = false;

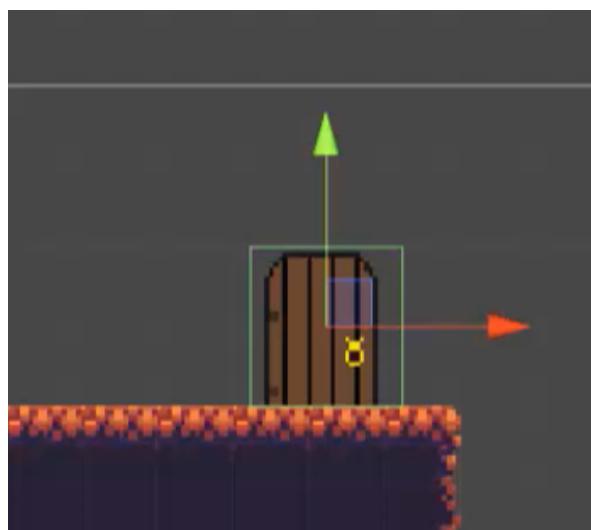
Nous avons à présent un joueur qui meurt parfaitement ainsi que l'animation associée.

6) Ajout de niveau ainsi que des transitions

Notre jeu est composé de 4 niveaux différents ainsi nous avons donc créé 4 scènes correspondant à chaque niveau avec des tilemaps et des fonds de scènes différentes afin d'avoir des décors propres à chaque niveau.

Ainsi nous avons donc dû mettre en place des transitions entre chaque niveau pour passer d'une scène à une autre.

Pour ce faire, nous avons ajouté le sprite d'une porte pour représenter la fin du niveau actuel et donc le début du niveau suivant.



Le principe est lorsque le joueur sera suffisamment proche de la porte c'est-à-dire lorsqu'il sera entré dans sa boîte de collision alors on active la transition avec le niveau suivant.

Nous avons créé une classe *LoadSpecificScene* pour gérer la transition en les niveaux: nous avons ajouté un namespace spécifique pour la gestion des transitions qui se nomme *UnityEngine.SceneManagement* ; nous avons utilisé la méthode *OnTriggerEnter2D* qui va permettre au joueur de changer de niveau dès qu'il sera en contact avec la porte.

```
4  public class LoadSpecificScene : MonoBehaviour
5  {
6      private void OnTriggerEnter2D(Collider2D collision)
7      {
8          if(collision.CompareTag("Player"))
9          {
10              SceneManager.LoadScene("Level02");
11          }
12      }
13 }
```

Cependant nous nous sommes rendus compte que lorsqu'il y a une transition vers le niveau suivant, la barre de vie du personnage se réinitialise au maximum même si le joueur avait perdu des points de vie dans le niveau précédent.

Pour garder les données du niveau précédent, nous avons utilisé la méthode *DontDestroyOnLoad* qui permet de conserver les données souhaitées ici la barre de vie dans la scène précédente afin de ne pas les supprimer dans la scène suivante.

```
public GameObject[] objects;

void Awake()
{
    foreach (var element in objects)
    {
        DontDestroyOnLoad(element);
    }
}
```

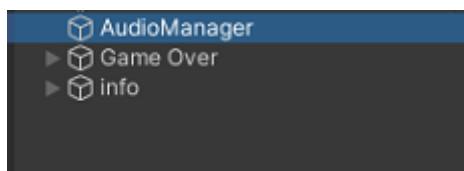
Enfin comme pour les zones d'élimination, nous avons introduit une animation de fondu pendant la transition d'un niveau à un autre (cf. Zone d'élimination).

E. Ingénieur Son et Réseau :

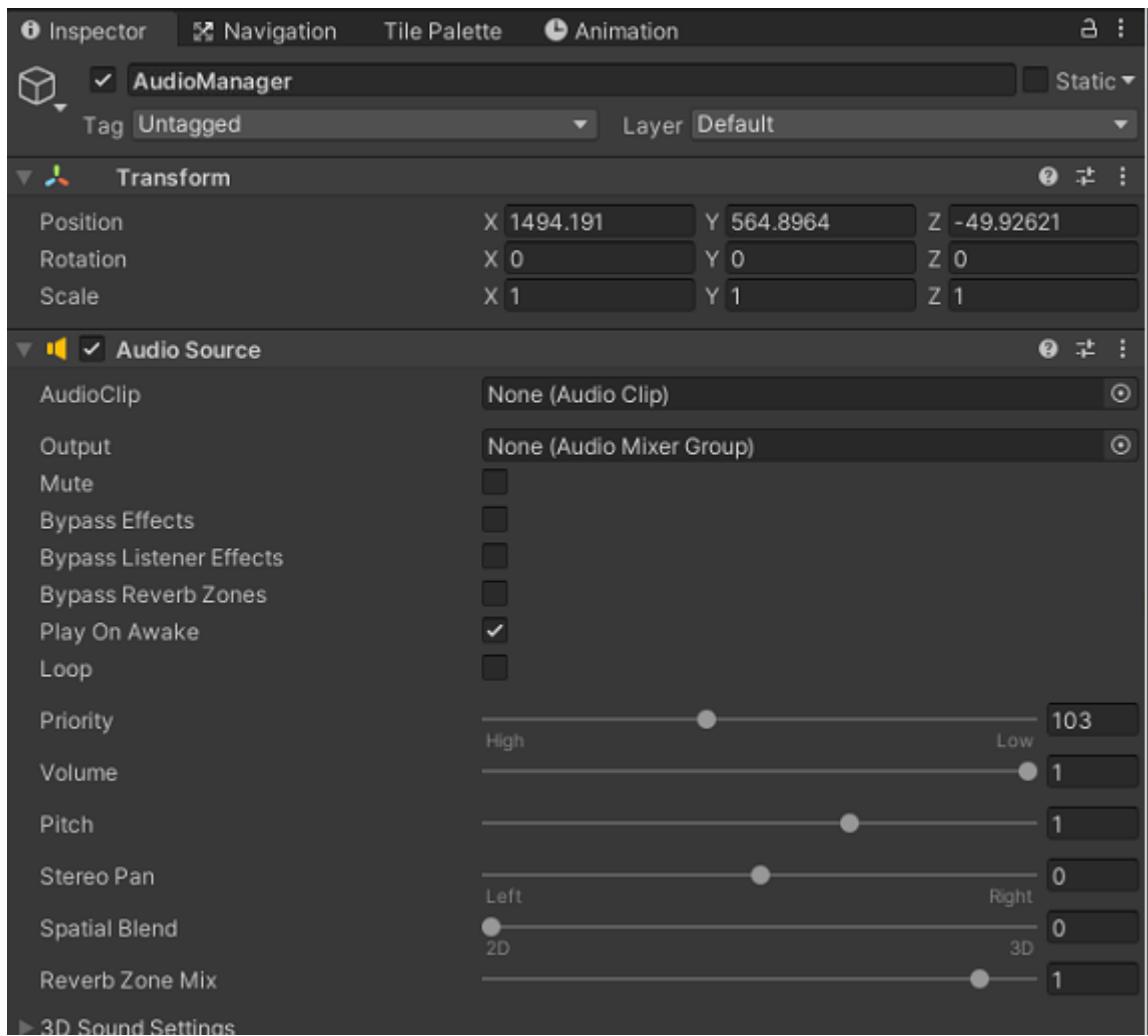
Pour cette partie, nous nous sommes procurées des musiques sans droit et gratuitement utilisables sur le site “Zapslat.com”, toutes trouvées et sélectionnées selon des mots clés qui correspondaient le mieux à nos niveaux.

Une fois le choix fait, nous avons dû les importer dans un dossier nommé “Audio/” se trouvant lui-même dans le dossier “Assets/” (dossier où toute les ressources du jeu se trouvent).

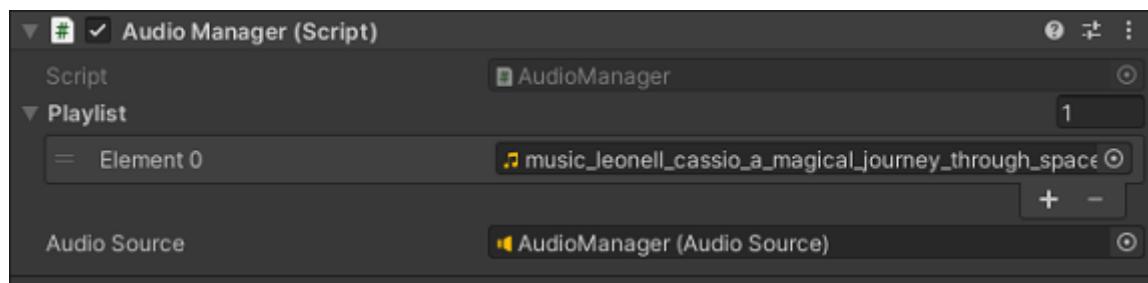
A partir de là, pour chaque niveau, nous allons créer un objet, cet objet aura pour composant, le composant “ AudioSource ” il fera donc tout naturellement office de source de son pour notre niveau.



Nous faisons bien attention à paramétrer la perception du bruit en 2D de sorte à ce que quel que soit l'endroit où l'on se trouve dans le niveau, notre perception de la musique reste la même (c'est-à-dire la même intensité et le même son).



Nous devons également ajouter un script AudioManager, qui nous permet de gérer une liste de musique, une liste que l'on va coder.



A l'intérieur de ce script on commence par implémenter la “playlist” de la classe AudioClip (liste de musique indexée) qui va passer en boucle les musiques qu’elle contient.

Enfin, nous avons implémenté également une source de la classe AudioSource. Dans la section *void Start()*, on initialise la source audio avec la première musique de la musique, puis on la joue avec la méthode *Play()*.

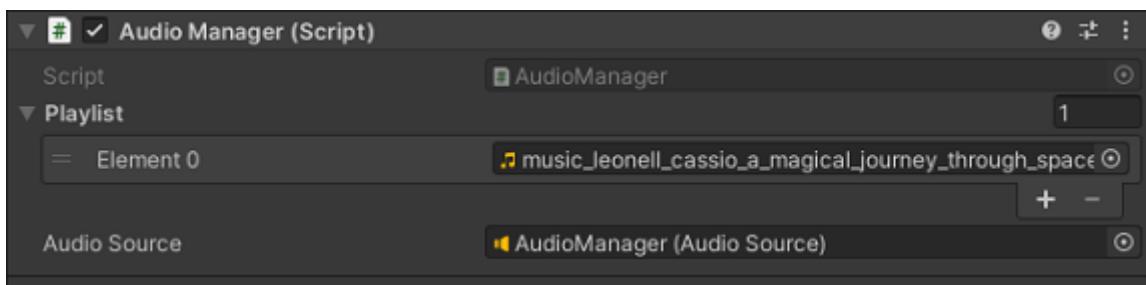
```
public AudioClip[] playlist;
public AudioSource audioSource;
```

Dans la partie *Update()*, nous allons relancer la musique si elle n'est pas en train d'être jouée en appelant la fonction *PlayAgainSong()*.

```
④ Message Unity | 0 références
void Start()
{
    audioSource.clip = playlist[0];
    audioSource.Play();
}

// Update is called once per frame
④ Message Unity | 0 références
void Update()
{
    if (!audioSource.isPlaying)
    {
        PlayAgainSong();
    }
}
1 référence
void PlayAgainSong()
{
    audioSource.clip = playlist[0];
    audioSource.Play();
}
```

Une fois cela fait, on retourne sur le moteur de jeu. Avec les musiques que nous avions importé dans le fichier “Audio/” et nous les mettons dans la composante AudioManager, dans la section playlist, ce qui va définir sa taille, permettant au code de s'adapter automatiquement à cette taille.



F.Le site web

D'une part pour la deuxième soutenance, nous avions fait des modifications de notre site internet: plus précisément nous avons complété les onglets vides notamment celui sur la présentation du projet dans lequel nous parlons du but du jeu et du synopsis pour permettre aux joueurs de mieux comprendre le concept et l'intérêt du jeu et encore de la présentation des membres du groupe avec nos photos.

De plus, nous avons mis à jour l'onglet *Archives* qui contient les différents rapports de soutenances en ajoutant le premier rapport de soutenance.

Enfin nous avons customisé notre site en modifiant la barre d'accueil et en ajoutant des fonds oranges sur les blocs de textes pour les rendre plus esthétique.

D'autre part, de la deuxième à la dernière soutenance, nous avons ajouté le précédent rapport de soutenance ainsi que celui-ci dans l'onglet *Archives*.



En outre, nous avons complété l'onglet *Ressources* avec tout ce qui nous a permis de faire le jeu: les bandes-sons pour accompagner le jeu et les différents logiciels pour les aspects artistique et technique du jeu.

Les différentes ressources que nous avons utilisées se trouvent ici :

Conception du jeu

Nous avons utilisé le logiciel "Unity" qui est une plateforme de développement qui nous permet de créer des jeux vidéos en 2D. Ce logiciel utilise du C# comme langage de programmation donc nous avons dû utiliser "Visual Studio" pour coder notre jeu.

Création du site Web et hébergement

En ce qui concerne le site web sur lequel vous vous trouvez actuellement, nous avons utilisé l'éditeur de texte "Sublime Text" : nous avons pu utiliser le HTML pour la structure et le CSS pour la décoration du site. Pour faire notre site, nous avons appris à coder grâce à différents tutoriels par exemple 3 min de Micode ou encore OpenClassroom. De plus notre site est hébergé sur la plateforme "GitHub" qui est gratuite et qui nous permet de faire des modifications directement sur notre site web .

Graphisme et Interface

Nous avons dû utiliser d'autres logiciels pour améliorer l'aspect graphique du jeu et pour construire les interfaces du jeu. Nous avons dessiné les différentes tilemaps autrement dits les décors à l'aide du logiciel "Aseprite". Pour l'interface les croquis des différents menus ont été réalisés sur "Photofiltre" avant d'être relié sur Unity

[Musique niveau 2](#) [Musique niveau 1](#) [Musique niveau 3](#) [Tiles niveau 1 & 3](#) [Tiles niveau 2](#)

Enfin, nous avons ajouté le lien pour le téléchargement du jeu dans le dernier onglet *Téléchargement du jeu* pour permettre à l'utilisateur de jouer. Le manuel d'installation sera également disponible sur cet onglet pour que le joueur puisse télécharger notre jeu sans difficulté.

III. Problèmes rencontrés

Au cours de notre projet, nous avons été confrontés à de nombreux problèmes et soucis techniques. La première cause de ces retards est due à notre manque d'expérience, en effet il s'agissait de notre tout premier projet informatique.

Le fait de travailler à distance nous obligeait à répartir les différentes tâches pour que chacun avance sur sa partie à partir de son propre ordinateur. Pour cela nous avons créé un dépôt Git via GitHub, cette méthode est très efficace car elle permet que tout le monde puisse travailler en même temps sans compromettre les possibles changements des autres participants au projet, malheureusement Git est difficile d'utilisation et il n'est pas très intuitif pour des novices comme nous même après visionnage de plusieurs vidéos et sites d'explication du fonctionnement de Git.



Cette méthode ne nous a pas posé de problèmes pour les deux premières soutenances même jusqu'à une semaine avant la soutenance finale, à partir de là, un bug ou une mauvaise manipulation de Git a corrompu les fichiers du jeu, à cause de cela, il était alors impossible de lancer le jeu. Nous avons alors essayé par différents moyens de récupérer l'ancienne version via les commandes de Git cependant les fichiers étaient toujours corrompus.

Nous avons dû trouver une autre solution, grâce à une ancienne version sauvegardée en local nous avons pu récupérer une certaine partie de

notre projet néanmoins une grande partie des avancées faites ont dû être refaite, pour cela dans un premier temps nous avons déposé cette sauvegarde dans un dossier Google Drive mais nous avons vite remarqué que cette solution n'était pas viable car nous perdions trop de temps à importer et récupérer les changements faits par chacun et les fichiers étaient malgré tout toujours corrompus.



Il nous restait alors peu de temps avant la soutenance finale, nous avons alors dû faire preuve de créativité, pour cela nous avons utilisé le logiciel TeamViewer qui permet de contrôler un ordinateur à distance et grâce à cela nous avons pu chacun notre tour refaire ce que nous avions perdu et avancer sur les derniers changements, même si cette solution n'était pas la plus efficace, elle nous a permis de finir dans les temps.



TeamViewer

VI. Conclusion:

Pour conclure cette soutenance finale nous sommes tout de même satisfait du résultat final. La plupart des objectifs que nous nous étions fixés ont été respectés. Nous sommes tout de même déçus de ne pas avoir implémenté la partie Réseaux à notre jeu pour le mode multijoueur car nous avons dû faire face à des difficultés: les problèmes rencontrés avec Git qui nous ont pris beaucoup de temps.

Concernant l'ambiance du groupe elle a toujours été très bonne du début jusqu'à la fin du projet malgré les difficultés liés au covid pour nous voir pendant la réalisation du projet.

Nous sommes tout de même heureux d'avoir réalisé ce premier jeu qui fut une première pour tous les membres du projet. Nous avons acquis de nombreuses connaissances que ce soit sur le plan technique avec la réalisation d'un jeu vidéo 2D ou même sur le plan oratoire avec les 3 soutenances. Nous en sortons mieux préparé pour l'année prochaine et pour la suite.

