

Projet Année

L3 Informatique

Ricquet Alicia
Marechal Baptiste
Larmat Jean Pierre



Sommaire

Présentation:	2
Projet	2
Moteur Unity	3
Planning	5
Conceptualisation:	6
Concept Globale:	6
Personnage jouable.....	8
Enemies.....	10
Éléments Visuels.....	13
Développement:	18
Personnage jouable.....	20
Camera.....	39
Enemies.....	43
<i>Squelette</i>	48
<i>Fantôme</i>	53
<i>Démon Volant</i>	56
<i>Squelette Violet</i>	59
<i>Démon Volant Rouge</i>	61
<i>Goblin</i>	63
Map.....	66
Conclusion:	69
Larmat Jean.....	69
Ricquer Alicia.....	69
Marechal Baptiste.....	70
Références:	70

Présentation

Projet

Pour notre projet de cette année, notre choix s'est porté sur le développement d'une démo de jeu vidéo. Notre concentration principale sera axée sur la programmation, englobant divers aspects tels que la structuration du code, l'environnement et structuration du jeu, la carte, la jouabilité, les personnages jouables, la physique du jeu, les mécaniques du jeu (règles, points de vie, capacités du joueur) et les ennemis, avec une base d'intelligence artificielle (conditionnel). Nous laisserons de côté les aspects graphiques en utilisant des textures déjà disponibles en libre droit.

La mise en place revêt une importance capitale, c'est pourquoi nous avons opté pour la méthode d'organisation professionnelle couramment utilisée dans l'industrie du jeu vidéo, le Game Development Document (GDD). Ce document détaillera les concepts, la composition du jeu, les personnages (jouables ou non), les ennemis, les obstacles, le contenu, les règles, le fonctionnement des physiques , ainsi que la manière dont chaque élément est construit, illustré par des diagrammes d'algorithmes de flux, des esquisses, des

Il est important de remarquer que, bien que le produit final soit une démo du jeu, notre conception sera élaborée comme si nous avions l'intention de terminer le jeu intégralement. Cette démo vise à mettre en lumière l'application de nos connaissances en programmation, ainsi que les compétences nouvellement acquises dans la gestion de projet et la création d'un multi média numérique interactif, à savoir un jeu vidéo.

Ainsi, dans ce compte rendu, nous commencerons par la conception du projet, basée sur un GDD où l'on met en place l'idéal du jeu vidéo tout en anticipant les défis liés au développement. Ensuite, après avoir structuré une idée cohérente et réalisable, nous passerons au développement, mettant en pratique ce que nous avons noté lors de la conception. Cette phase implique beaucoup de communication pour faciliter les interactions entre les différentes parties sur lesquelles chacun travaille. Il est essentiel de s'assurer que personne ne reste en arrière et que les dépendances du projet soient gérées simultanément (par exemple, l'interaction entre l'attaque du personnage et la réaction de l'ennemi à l'attaque). Enfin, nous présenterons les résultats obtenus et analyserons notre parcours, ce que nous avons appris et ce que nous pourrions améliorer.

Objectives:

Nous avons entrepris la création d'une démo de jeu rétro, axée sur la difficulté, dotée d'un gameplay rapide et dynamique, mais avant tout, conçue pour offrir une expérience amusante au joueur. Notre intention est de produire une certaine nostalgie en rappelant les jeux d'arcade classiques. Nous nous inspirons notamment de titres emblématiques tels que Megaman et Castlevania. Tout en puisant dans ces références rétro, nous intégrons des éléments physiques plus contemporains et dynamiques, explorant une possible inspiration

tirée de jeux modernes tels que Hollow Knight. L'objectif est de créer une fusion entre la réminiscence des jeux classiques et des mécaniques de jeux actuelles, créant ainsi une expérience à la fois familière et novatrice pour le joueur.

Enjeux:

Conception de Niveaux Équilibrée : Créer des niveaux équilibrés et engageants dans un Metroidvania peut être complexe. Il faut s'assurer que la progression du joueur est fluide, tout en offrant des défis stimulants et en maintenant un équilibre entre l'exploration et l'action.

Carte du Monde Cohérente : La conception de la carte du monde dans un Metroidvania est cruciale. Assurez-vous que la disposition des zones, la connectivité des niveaux, et la distribution des compétences et des objets permettent une exploration intéressante sans frustration.

Mécaniques de Combat Équilibrées : Équilibrer les mécaniques de combat pour que le joueur se sente puissant mais que les ennemis restent un défi est essentiel.

Récompenses et Rétroaction : Trouver un équilibre dans la distribution des récompenses et des power-ups est crucial pour maintenir l'intérêt du joueur. La rétroaction visuelle et auditive doit également être claire pour indiquer au joueur le succès ou l'échec.

Narration Intégrée : Intégrer une narration cohérente dans un jeu rétro peut être un défi vu que la narration dans les Metroidvania doit souvent être environnementale, et le monde lui-même doit raconter une histoire. Ceci parce que le jeu va se concentrer sur le gameplay plutôt que la narration.

Réflexion sur l'Accessibilité : S'assurer que le jeu est accessible à différents niveaux de compétence.

Références sans Imitation Directe : Créer un jeu qui s'inspire des classiques sans simplement les imiter peut être un défi créatif. Trouver une identité unique pour le jeu tout en honorant les éléments appréciés des jeux Metroidvania est important.

Tests et Rétroaction : Les phases de test sont cruciales pour s'assurer que la difficulté est équilibrée, que les bugs sont résolus, et que l'expérience du joueur est optimale. Obtenir des retours de joueurs est également important.

Limites :

Ne surchargez pas le jeu avec des objets qui n'ont rien à voir avec le type platformer, ni avec des mécaniques qui pourraient perturber les joueurs. Restez fidèle à un seul public cible : les amateurs de jeux rétro, les passionnés de défis, les joueurs à petit budget et les nostalgiques. Il n'est pas judicieux d'essayer de plaire à tout le monde. De plus, veillez à ce que le jeu ne dépasse pas un certain niveau de difficulté.

Moteur de jeu Unity

Unity est une plateforme de développement de jeux vidéo largement utilisée par les développeurs du monde entier. À la base, Unity est un moteur de jeu, offrant un ensemble d'outils puissants pour créer des expériences interactives et immersives.

Un moteur de jeux, souvent désigné sous le terme anglais "game engine", est un logiciel conçu pour simplifier et accélérer le processus de développement de jeux vidéo. Les moteurs de jeux fournissent généralement des fonctionnalités de base telles que la gestion des graphiques, de la physique, de l'audio, des entrées utilisateur, de l'intelligence artificielle et des animations. Cela permet aux développeurs de se concentrer sur la conception du jeu lui-même plutôt que sur le développement de systèmes fondamentaux.

Il offre un ensemble d'outils et de fonctionnalités prédéfinies permettant aux développeurs de créer des jeux interactifs sans avoir à construire chaque aspect du jeu à partir de zéro.

En outre, les moteurs de jeux offrent souvent des outils de création de contenu, tels que des éditeurs de niveaux, des systèmes de gestion de ressources et des interfaces de programmation d'applications (API) pour permettre aux développeurs de personnaliser et d'étendre les fonctionnalités du moteur selon leurs besoins spécifiques.

Comment ça fonctionne ?

Au cœur de Unity se trouve le concept de GameObjects, des éléments fondamentaux qui composent le monde du jeu. Ces GameObjects peuvent être des objets physiques, des personnages, des décors ou tout autre élément interactif. Ils sont enrichis par l'ajout de composantes, qui définissent leur comportement, leur apparence et leurs interactions avec l'environnement du jeu.

Les scénarios, ou scènes, sont des environnements dans lesquels ces GameObjects interagissent. Les développeurs utilisent Unity pour concevoir ces scènes en plaçant et en configurant les GameObjects, en définissant les interactions et les règles du jeu.

Pour contrôler le comportement des GameObjects et créer des interactions dynamiques, les développeurs utilisent des scripts écrits en C#. C# est un langage de programmation Orienté Objet inspiré de Java, principalement utilisé pour le développement d'applications, qui offre une syntaxe claire et concise. Dans Unity, les scripts en C# sont attachés aux GameObjects et sont exécutés à différents moments pendant le déroulement du jeu, permettant ainsi aux développeurs de contrôler précisément le comportement de leurs jeux.

Unity facilite le développement de jeux vidéo en offrant une interface conviviale et intuitive, ainsi qu'une vaste gamme de fonctionnalités prêtes à l'emploi, telles que la physique, les graphismes avancés, la gestion des animations, le son, la gestion des entrées utilisateur et bien plus encore. De plus, Unity prend en charge plusieurs plateformes de déploiement,

permettant aux développeurs de créer des jeux pour PC, consoles, appareils mobiles et réalité virtuelle, le tout à partir d'un seul projet.

En résumé, Unity est bien plus qu'un simple moteur de jeu. C'est une plateforme complète qui offre aux développeurs les outils nécessaires pour donner vie à leurs idées de jeux vidéo de manière efficace et professionnelle, tout en leur permettant d'exprimer leur créativité à travers le langage de programmation C#.

Planning

Par les contraintes de temp , les trois partie principales dans lequel en va le plus travailler vont être:

- Le personnage jouable, fait par Marechal Baptiste
- Le map, plateformes, fait par Riquet Alicia
- Les ennemis , fait par Larmat Jean

Tâches	Duree	Date debut	Date fin	Dependence	En charge
Initiation					
1. Identifier le type de jeu					Tous
2. Identifier objectives et enjeux du projet					Tous
3. Identifier perimetre				2	Tous
4. Identifier public visé				2	Tous
5. Planning				4	Jean Pierre
Conceptualisation					
6. Determiner Concept du Jeu					Jean Pierre
7. Créer Pitch				6	Jean Pierre
8. Monde du jeux					Tous
9. Determiner objectives du jeu				6	Alicia
10. Determiner Progression du jeu				6	Jean Pierre
11. Personnage jouable				9, 10, 15	Baptiste
12. Environnements et niveaux				9, 10 , 15	Alicia
13. Ennemis				9, 10, 15	Jean Pierre
14. Camera					Baptiste
15. Element Visuels					Tous
Developpement					
Prototypes					Tous
16. Personnage jouable					Baptiste
17. Ennemis					Jean Pierre
19. Niveaux					Alicia
20. Verification des prototypes					Tous
21. Correction ou addition fonctionnalites					Tous
Finalisation					
22. Test & Debug					Tous
23. Enregistrement des videos					Tous

Conceptualisation

Concept Globale

Titre du jeu: “A Will to Live Again”

“A Will to Live Again” est un jeu de plates-formes d'action aventure en 2D du type metroidvania. Se déroulant dans un univers de Dark Fantasy où le joueur prend le rôle d'un malheureux aventurier très agile et déterminé à s'infiltrer dans le château d'un être supérieur démoniaque pour le neutraliser et mettre fin à son royaume de terreur. Le joueur doit traverser différents scénarios dynamiques dans les alentours ou l'intérieur d'un château maudit en ruines où il rencontre des structures qui bougent pour l'obstruer et le mettre dans des situations compromettantes, des pièges mortels cachés ou à l'attente du joueur et des ennemis, des fois au sol, des fois en l'air, qui sont très agiles ou très puissants parmi autres. De plus, la progression peut exiger de vaincre des hordes d'ennemis pour ouvrir son chemin et continuer le parcours. Chaque étage est ainsi couronné à la fin par un mini boss redoutable, testant ainsi la réactivité du joueur et ces compétences acquises sur le système de combat et les mécaniques au fur et à mesure du jeu.

Les ennemis sont aussi bien élémentaires que complexes. Des ennemis avec de mouvement simple qui gardent uniquement une zone et ne frappent que tout ce qui se présente sur leur chemin, aux bosses qui sont capables d'esquiver vos attaques, d'utiliser différents pouvoirs et mouvements et sont résistants à vos attaques pour subir plus de dégâts. Néanmoins, en explorant le monde et en trouvant des coffres ou en éliminant des ennemis, le personnage principal a l'opportunité d'acquérir des nouveaux équipements qu'il peut garder dans son petit équipage. Ces équipements sont principalement des différents types d'armes qui changent sa façon de combat et des outils comme des potions qui l'aident à se soigner ou augmenter ses capacités physiques, offrant finalement au joueur de l'adaptabilité et des options stratégiques pour faire face aux défis exigeants et triompher des étages les plus redoutables. “A will to Live again” offre ainsi une expérience stimulante, combinant des mécaniques de jeu dynamiques et actives avec une variété d'ennemis effrayants et des défis difficiles, tout en offrant à la fin au joueur une progression satisfaisante à travers les victoires des défis.

Pitch:

“A Will to Live Again” est un jeu de plate-formes d'action frénétique mettant au défi les joueurs avec un gameplay rapide et exigeant. Plongez dans une mésaventure où tu devras affronter dès terrains dangereux avec des obstacles vivants qui essaieront de vous faire reculer bien que de vous tuer en même temps que vous devrez faire face à des ennemis, des fois en horde, des fois pendant son trajet, des fois pour pouvoir accomplir l'étage. Pourtant vous ne serez pas condamné vous pourrez trouver dans votre parcours des objets comme des d'armes ou des potions qui pourront faciliter vos affrontements aux différentes péripéties. Ainsi vous mettrez vos compétences des reflets et de versatilité pour survivre

aux terrifiants affrontements qui vous seront mis en place. N'ayez pas peur, et mets toi dans une petite odyssée obscure, riche en défis et en dangers ou votre détermination sera mise à l'épreuve à chaque seconde et ou le triomphe sera très satisfaisant.

Progression:

La progression du joueur dans le jeu repose sur un équilibre délicat entre linéarité et liberté. Dans un Metroidvania, il est crucial de créer un monde ouvert et non linéaire qui encourage l'exploration tout en offrant des défis adaptés au niveau de compétence du joueur. Dans ce contexte, la gestion de la progression du joueur devient essentielle. Le jeu est structuré autour de l'idée d'avancer à travers différents niveaux pour devenir plus fort et finalement vaincre le roi démon. Chaque niveau présente des défis uniques et une difficulté progressive, avec l'apparition de monstres démons de plus en plus redoutables au fur et à mesure que le joueur progresse. Cette progression est accompagnée de la possibilité pour le personnage de gagner en puissance grâce à l'acquisition de nouveaux équipements, d'armes plus puissantes et de potions pour restaurer sa santé. Pour garantir une expérience de jeu fluide, des checkpoints sont placés stratégiquement dans les niveaux, permettant au joueur de sauvegarder sa progression et de reprendre là où il s'était arrêté en cas de défaite ou de sortie du jeu. Cette approche offre au joueur un sentiment de satisfaction en accomplissant des défis tout en maintenant un niveau de difficulté stimulant et gratifiant.

Conception personnage jouable

Le personnage principal est contrôlé par le joueur et interagit avec le monde du jeu.

Mouvements :

- Déplacements de gauche à droite Le personnage peut se déplacer horizontalement, de gauche à droite à travers le niveau du jeu.
 - saut :Le personnage peut effectuer un saut pour franchir des obstacles, atteindre des plates-formes élevées, ou éviter les dangers.Il n'est pas possible de sauter sans toucher le sol .Plus le joueur reste appuyer sur la touche de saut plus le personnage sautera haut.
 - gravité : La gravité affecte le mouvement du personnage en le faisant retomber après un saut ou pendant une chute.
 - roulade : Le personnage peut effectuer une roulade pour se déplacer rapidement, esquiver les ennemis ou éviter les pièges.
-
- s'accroupir/chute rapide : en appuyant sur la direction vers le bas en étant en l'air la gravité augmente et le personnage chute plus rapidement. Quand il est au sol et qu'il appuie sur la direction basse le personnage est accroupi cela ne change presque rien a part que le joueur ne peut pas sauter en étant accroupi.

Collisions :

- Collisions avec le terrain : Le personnage peut entrer en collision avec le sol, les murs et d'autres éléments du décor
- Collisions avec un ennemi ou une attaque d'un ennemi : Si le personnage entre en collision avec un ennemi ou subit une attaque ennemie, cela entraîne une perte de points de vie et le personnage est repoussé et ne peut pas bouger pendant qu'il se fait repousser.

Attaques :

- dégâts sur les ennemis : à la suite d'une attaque qui touche un ennemi, celui-ci perd de la vie.
- épée : permet d'attaquer les ennemis à courte porté
- arc : permet de tirer une flèche qui inflige des dégâts à un ennemi.

Points de vie :

- Le personnage commence avec 10 points de vie
- affichage sur le HUD (heads-up display) : Les points de vie du personnage sont affichés à l'écran dans l'interface utilisateur et sont mis à jour quand les points de vie changent.

- perte de points de vie : Lorsque le personnage subit des dégâts, ses points de vie diminuent.
- Game Over : Si les points de vie du personnage atteignent zéro, le jeu affiche un écran de "Game Over" et le joueur peut être amené à recommencer

Caméra :

- Suivre le joueur (avec délai) : La caméra suit le personnage principal tout en ajoutant un léger délai pour créer un effet de fluidité dans le suivi.
- bloquée par les murs : La caméra ne peut pas traverser les murs
- avancée dans la direction du regard du personnage principal : La caméra suit le regard du personnage, montrant ce qu'il se dirige vers.
- plus grand délai de déplacement sur le début du saut que pendant la chute : Le délai de déplacement de la caméra peut varier en fonction de la situation, avec un délai plus important au début d'un saut pour mieux suivre le personnage en l'air.
- salles où elle est fixe : Dans certaines salles ou zones du jeu, la caméra peut être fixe et ne pas suivre le personnage, ce qui peut ajouter de la variété aux perspectives de jeu.

Conception enemies:

Dans le parcours du jeu, outre la structure de la carte, les plateformes mobiles et les pièges, ce qui sera également essentiel et offrira des défis et des difficultés au joueur seront les ennemis qu'il rencontrera tout au long du jeu. Ces ennemis seront caractérisés par différents niveaux de difficulté, allant des ennemis faciles avec des mouvements et des schémas simples, tels que la patrouille dans une petite zone avec une vitesse réduite et une faible quantité de vie, à des mouvements plus dynamiques, rapides et complexes, tels que la poursuite du joueur d'une plateforme à une autre et leur capacité à subir plus de dégâts. Nous verrons que nous avons dû prendre en compte de nombreux détails en travaillant avec les composants physiques et game objects d'unity. Au début ils peuvent passer inaperçus mais plus on développe le code du PNJ (personnage non jouable) on découvre ces détails qu'il faut prendre en compte pour éviter les bugs et des nouvelles fonctionnalités de Unity à apprendre.

En plus des mécaniques et des difficultés programmées des ennemis, leur emplacement dans la carte joue un rôle crucial pour rendre le jeu à la fois amusant et stimulant. Le modèle de la carte, des plateformes et des pièges, ainsi que l'emplacement des différents ennemis, influent sur leur niveau de difficulté. En effet, par rapport à la carte et aux plateformes elles-mêmes, même les ennemis de bas niveau peuvent présenter une grande difficulté, et vice versa. L'idée serait que, une fois que le joueur a découvert tous les ennemis existants dans une zone et s'est habitué à leurs mécanismes, on les mélange pour créer des combinaisons d'ennemis originales. Par exemple, en mélangeant des ennemis avec différents niveaux de difficulté, les ennemis basiques peuvent servir d'appui pour rendre plus difficiles les ennemis de haut niveau. Ces combinaisons auraient pour objectif de présenter de nouveaux défis et de surprendre le joueur, évitant ainsi l'ennui.

Ainsi L'idée générale serait d'augmenter progressivement le niveau de difficulté des ennemis en fonction de l'avancement du joueur dans le jeu . Au départ, dans les premiers niveaux, seuls les ennemis basiques apparaissent. Ensuite, des ennemis de difficulté plus grande font leur apparition, et parfois même un boss pour marquer la fin d'un étage du jeu. A la fin on procurerait au joueur une sensation d'amélioration et de satisfaction en complétant des niveaux de plus en plus difficiles.

Ennemis basic

-Squelette: Le squelette est l'ennemi le plus répandu dans le jeu. Il attaque le joueur à courte portée s'il s'approche trop près. Sa vitesse est modérée, et il garde généralement des zones avec un mouvement de balancement de gauche à droite. Il possède un seul point de vie, rendant son élimination relativement facile pour les joueurs.

-Fantôme: Le Fantôme est une entité insaisissable qui suit le joueur dès que le joueur rentre dans son sonar. Sa vitesse augmente progressivement et il est

capable de traverser les structures. Il garde des zones avec un mouvement parabolique. Bien qu'il ne possède qu'une seule vie, il se ranime après quelques secondes d'avoir été tué, poursuivant ainsi toujours le joueur et le forçant à échapper à sa zone de poursuite. Il a 1 point de vie, rendant son élimination facile pour les joueurs.

Ennemis intermédiaire

-Squelette violet: Le squelette violet est une variation du premier squelette. Il est plus rapide, et garde une zone plus grande. Il a un champ de vision pas trop grand et dès que le joueur rentre dans son champ de vision, le Squelette va sprinter vers la position où il a aperçu le joueur pour attaquer. Il possède 2 points de vie, rendant son élimination relativement plus difficile que le squelette normal. De plus, quand il est tué, il existe une chance que son corps fasse apparaître un fantôme obligeant le joueur à réfléchir deux fois s'il vaut mieux tuer ou non l'ennemi, et si l'éviter serait la meilleure option.

-Démon Volant: Le Démon Volant est un ennemi aérien redoutable qui lance des projectiles et peut esquiver les attaques venant d'en dessous de lui. Il occupe des endroits où le joueur doit exploiter ses capacités d'agilité pour le vaincre, que ce soit à courte ou longue portée (horizontalement). Ses mouvements de droite à gauche occasionnels à vitesse intermédiaire en font un défi constant pour les joueurs.

-Démon Volant rouge: Démon Volant rouge est une variation du Démon Volant. Il apparaît en hauteur et attaque rapidement le joueur à courte portée avant de retourner à sa position d'origine. Il possède une attaque avec un temps de recharge, ce qui la rend difficile à affronter. Avec ses deux points de vie, elle nécessite une attention rapide et des réflexes vifs de la part du joueur pour arriver à trouver le bon moment pour attaquer.

-Gobelins: Le Gobelins est un ennemi terrestre rapide. Il poursuit activement le joueur dès que celui-ci entre dans son champ de vision, et il ne s'arrête jamais jusqu'à ce qu'il soit tué. Agile, le gobelin est même capable de suivre le joueur lorsqu'il saute d'une plateforme à une autre. Lorsque le joueur s'approche trop près, il engage l'attaque. Son attaque est à courte portée avec un temps de recharge, et il possède également deux points de vie, ce qui en fait un défi pour les joueurs moins expérimentés.

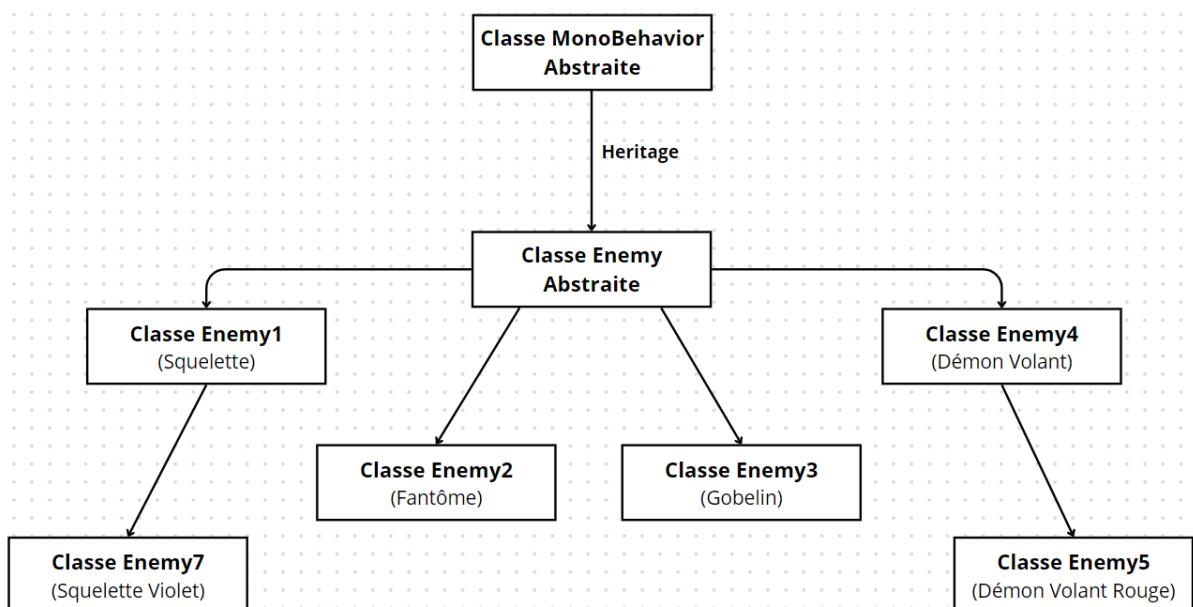
Ennemis avancé

//“théorie” malheureusement pas le temp pour l'implémenter //

-Boss: Le Boss présente une variété d'attaques, à la fois à courte et longue portée, avec une vitesse intermédiaire et la capacité de sauter. Il peut esquiver les attaques à distance et passer en mode défensif, obligeant le joueur à trouver son point vulnérable pour infliger des dégâts. Il possède un pouvoir dévastateur d'insta kill, ce qui en fait un adversaire redoutable nécessitant une stratégie soigneusement élaborée pour être vaincu.

Structure du code

Comme expliqué précédemment, en Unity, toute la partie programmation se fait en langage C#, un langage de programmation orienté objet. D'où mon idée principale pour la structure de base du codage des ennemis consisterait à utiliser de l'abstraction et l'héritage.



Classe Abstraite enemy:

Cette classe sera le pilier des ennemis, où nous écrirons les attributs et les méthodes de base que généralement tous les ennemis utiliseront. Ces méthodes peuvent être soit abstraites, soit virtuelles. Le terme abstrait signifie que c'est le fils qui implémente un corps pour la méthode, tandis que le terme virtuel signifie que le fils aura la possibilité de réécrire le corps de la méthode si nécessaire.

Tous les ennemis, vont avoir la structure de base suivante:

-Un script Contrôleur personnalisé, ceci sera le script principale ou nous allons gérer le différentes actions spécifique à l'ennemi, ces mouvement, c'est statuts (par exemple, si il est dans le sol, si il est mort, s'il a été touché par le joueur), c'est cooldowns ("temps de recharge") pour attaquer , pour se détruire etc...

- Un script manager d'interactions, pour gérer les interactions avec le map et le joueur, pour gérer si l'ennemi touche le sol, le joueur , ou le joueur attaque l'ennemi.

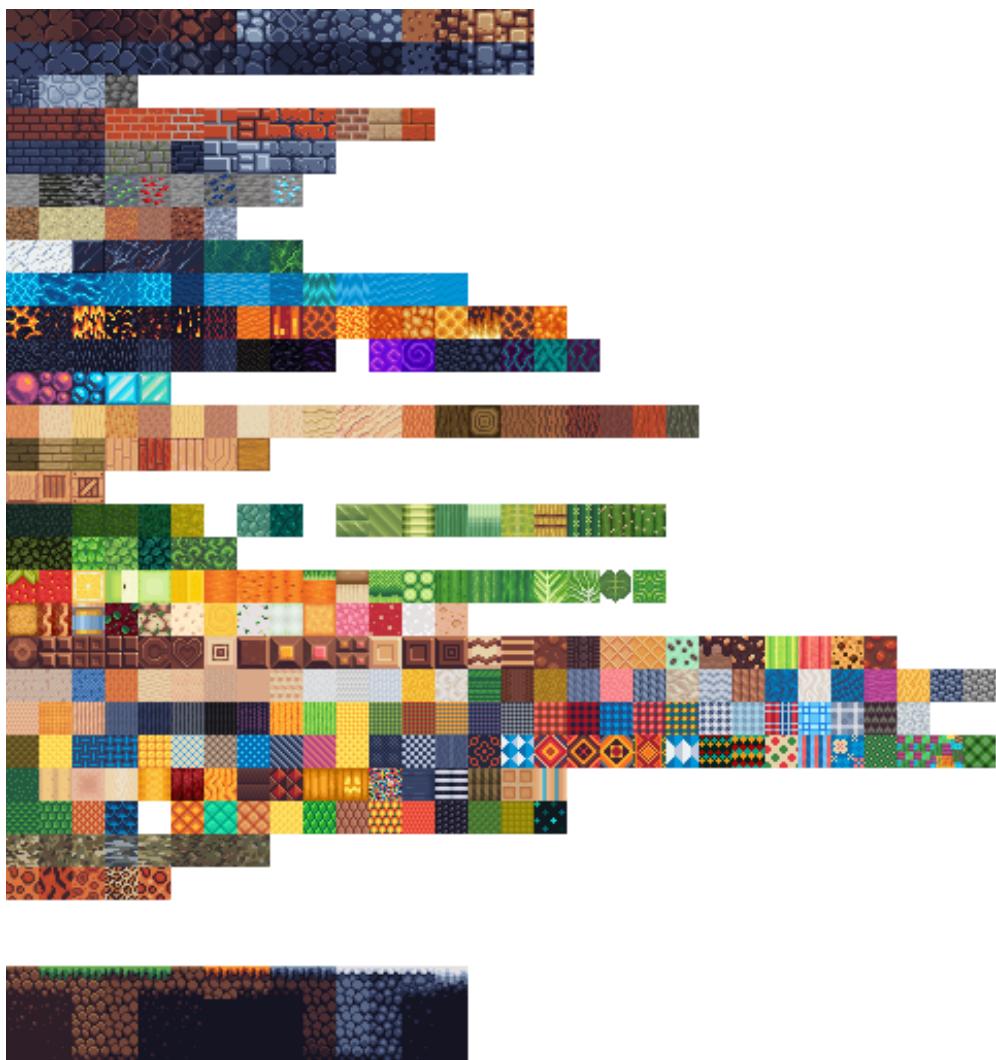
-Des points de vie.

-Une façon d'attaquer personnalisée

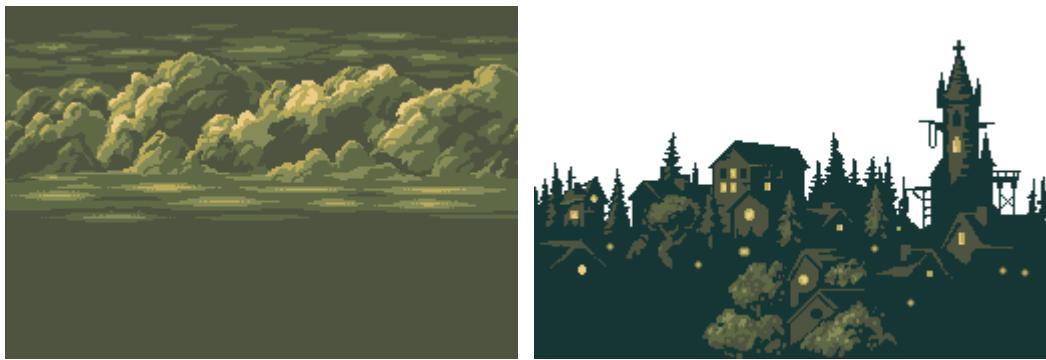
Après avoir annoncé les points les plus importants, évoqué l'idée générale de chaque ennemi ainsi que leur croquis, nous allons maintenant passer à la phase de développement.

Element Visuels:

Tiles:



Arrière-plan:



Personnage jouable:

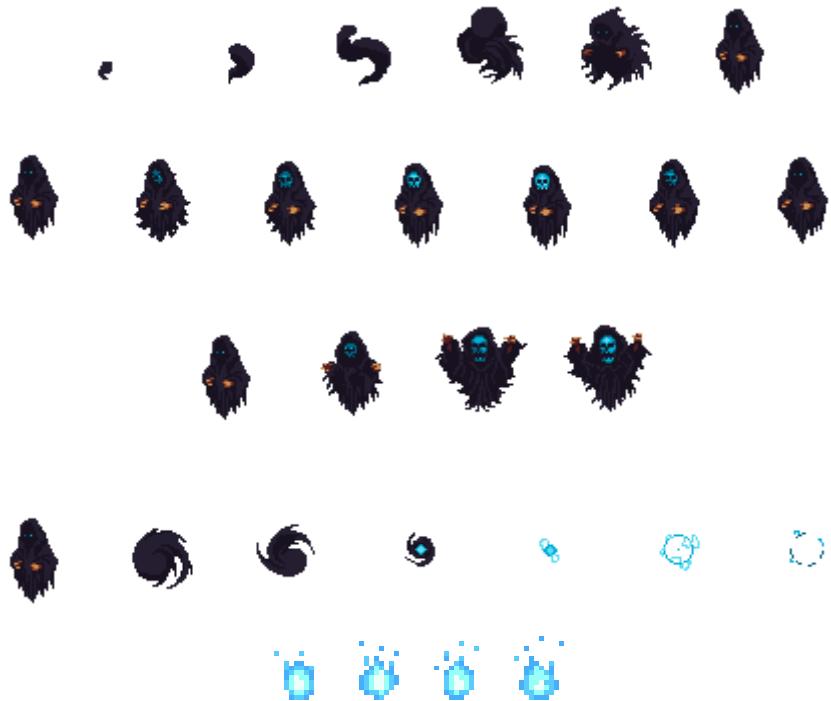


Enemies:

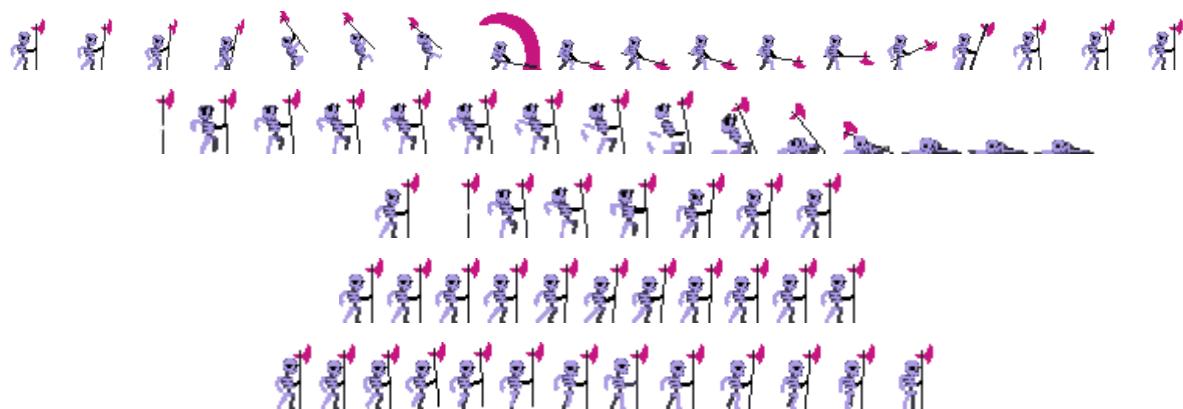
Skeleton:



Fantome:



Squelette Violet:

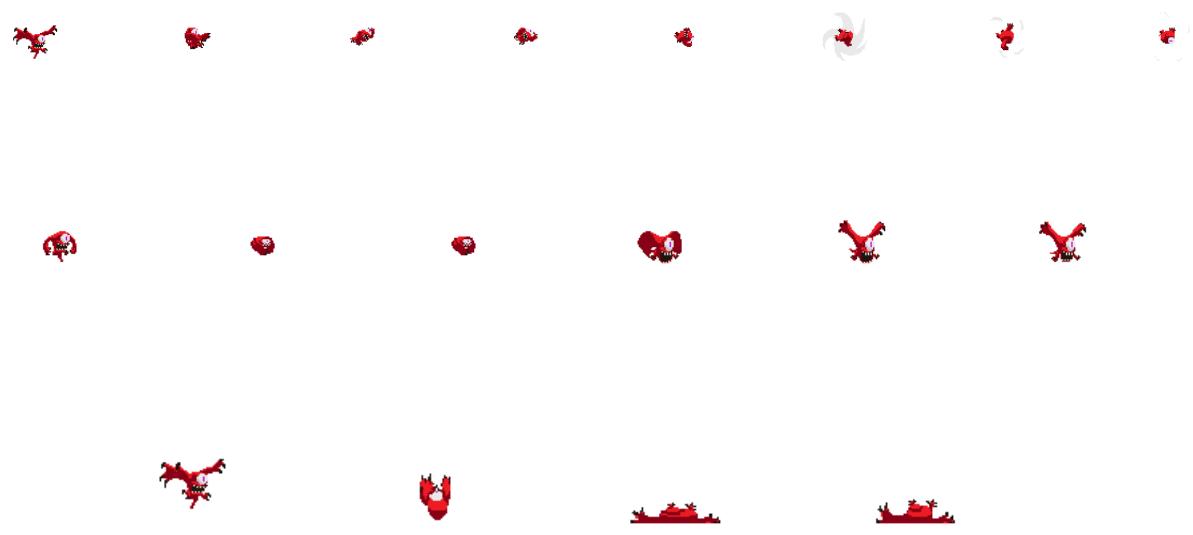


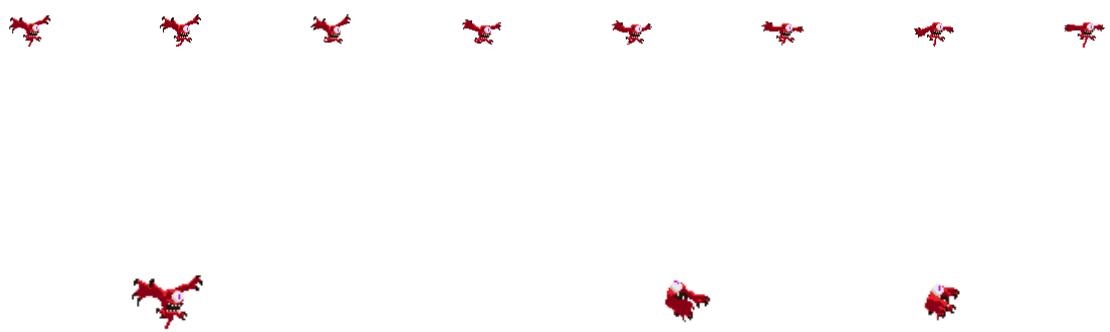
Démon Volant:





Démon Volant Rouge:





Gobelins:





Développement

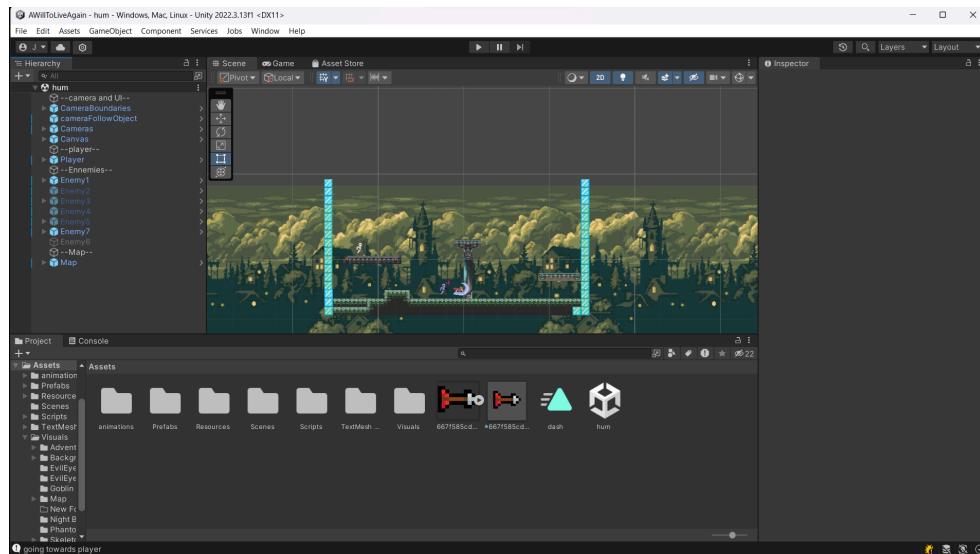
Après avoir chacun de nous défini nos vision des partie à faire, chacun va commencer a apprendre comment utiliser unity pour obtenir les resultat recherche. Le développement d'un projet sur Unity requiert une collaboration étroite entre les membres de l'équipe pour garantir la cohérence et l'efficacité du processus. Dans notre cas, Baptiste prendra en charge la construction du personnage principal, Alicia se concentrera sur la conception de la carte, tandis que Jean Pierre sera responsable de la création des ennemis. Cette répartition des tâches permet à chaque membre de se spécialiser dans son domaine tout en contribuant à l'ensemble du projet.

Cependant, dans un environnement de développement collaboratif, il est inévitable que certaines fonctions dépendent les unes des autres. Lorsqu'une fonction dépend d'une autre qui n'est pas de sa responsabilité, il est crucial que les membres de l'équipe se réunissent pour coordonner et synchroniser ces éléments. Cette approche favorise une intégration des différentes composantes du jeu et garantit que toutes les fonctionnalités s'imbriquent correctement.

Pour notre projet sur Unity, nous avons opté pour une base en 2D, ce qui permettra de concentrer nos efforts sur la création d'un environnement visuellement attrayant et immersif. De plus, nous avons choisi d'utiliser des physiques simples en faisant appel aux RigidBodies. Ces composants fournissent une simulation de la physique réaliste et simplifiée, ce qui facilite la manipulation et le mouvement des objets dans le jeu, tout en offrant une expérience de jeu fluide et convaincante.

Alors notre approche de développement sur Unity repose sur une répartition claire des tâches, une collaboration étroite entre les membres de l'équipe et l'utilisation de fonctionnalités telles que les Rigid Bodies pour créer un jeu 2D cohérent et captivant.

L'interface unity



Voici la fenêtre de unity elle est composée de plusieurs éléments :

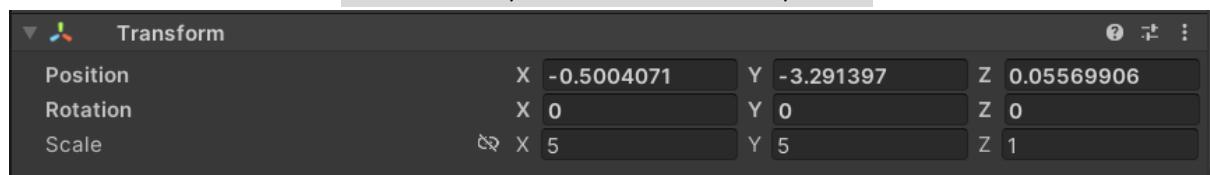
- à droite l'inspector qui sert à avoir un aperçu des éléments et des différents composants qu'il possède
- en bas la fenêtre de projet qui affiche les fichiers contenu dans le projet
- en haut à gauche les différents gameobject se trouvant dans la scène
- et au centre la scène là où l'on voit les gameobjects affichés et où on peut les placer comme on le souhaite

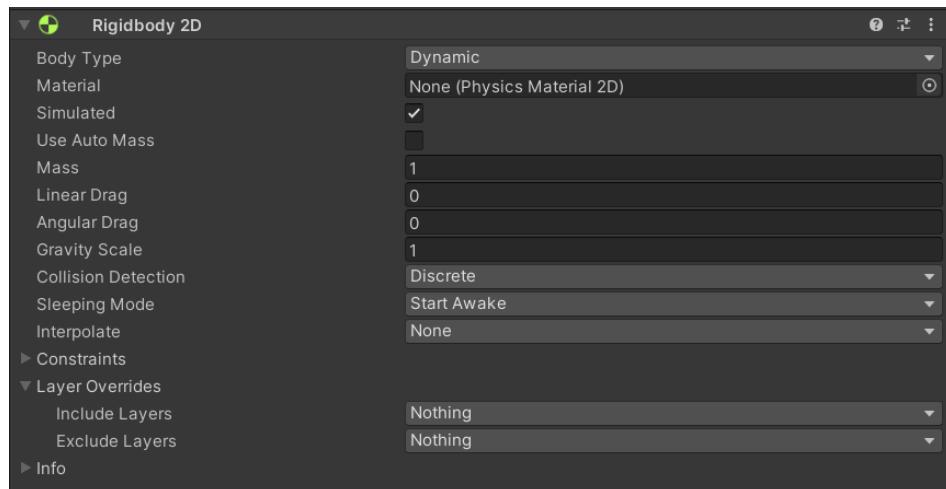
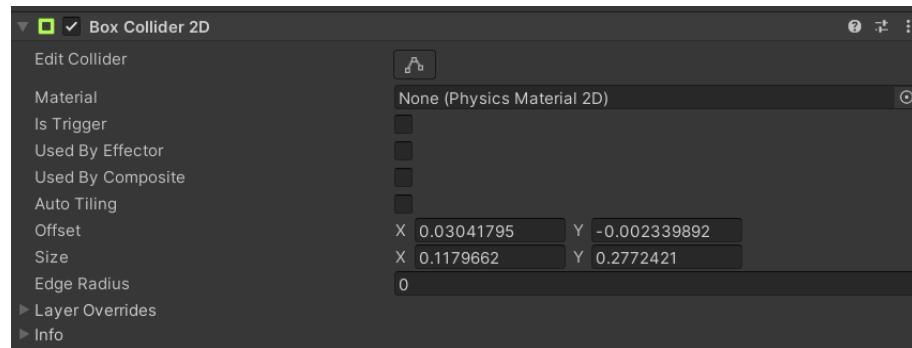
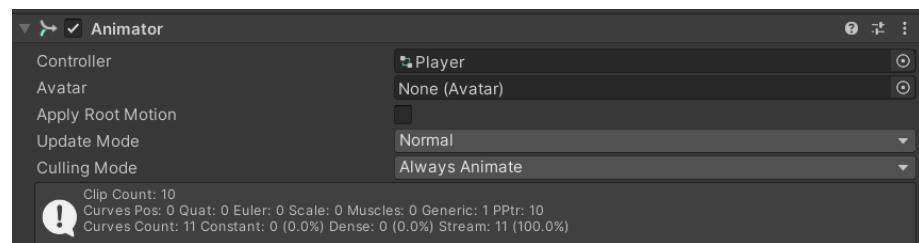
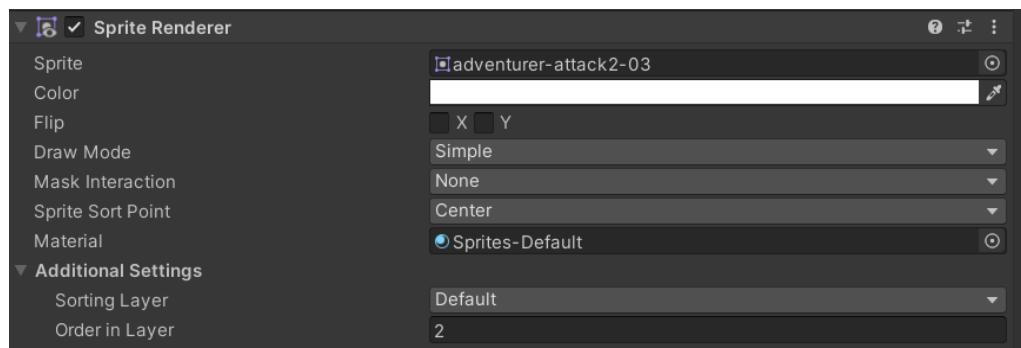
Personnage principal:

Pour commencer à créer le personnage principal il faut en premier lieu créer le gameobject et lui ajouter les composants nécessaires en suivant les choix de conception particulièrement la présence de gravité et de collision. On aura donc besoin des composantes basiques suivantes:

- Transform
- Collider2D (Box)
- RigidBody2D
- Sprite Renderer
- Animator

Voici en Unity dans la fenêtre “Inspector”:





Les GameObjects sont remplies par diverses composantes qui gèrent leur comportement, leur apparence et leurs interactions. Parmi ces composantes, nous trouvons

le Transform, qui détermine la position, la rotation et la taille d'un GameObject dans l'espace tridimensionnel du jeu. Grâce au Transform, les développeurs peuvent placer et orienter les objets de manière précise dans la scène.

Pour gérer les collisions et les interactions physiques dans le jeu, Unity offre plusieurs types de colliders, tels que les BoxCollider2D. Ces composantes permettent de définir des zones de collision autour des GameObjects, pour un box collider ce sera une zone rectangulaire, facilitant ainsi la détection des collisions avec d'autres objets et la gestion des interactions physiques.

Le Rigidbody est une composante essentielle pour simuler les comportements physiques réalistes des objets dans le jeu, puisque nous sommes sur un projet en 2D nous utiliserons des rigidbody2D. En attachant un Rigidbody à un GameObject, celui-ci devient sujet aux lois de la physique, comme la gravité et les forces externes, ce qui lui permet de se déplacer, de tomber, ou de réagir aux collisions de manière réaliste.

Sur unity on a la possibilité de modifier les paramètres des différents composants d'un gameobject pour un rigidbody, un des paramètre est important le body type qui peut prendre trois valeurs: dynamic, kinematic ou static

- Un Rigidbody static n'a pas de physique simulée on l'utilisera surtout quand un objet doit arrêter d'être simuler mais que l'on a besoin d'un Rigidbody le reste du temps. Ou juste sur un gameobject pour lequel on ne veut pas que le joueur soit affecté par les collisions avec ce gameobject.
- Un Rigidbody kinematic a des déplacements uniquement gérés par d'autres scripts et ignorent les physiques avec des gameobjects n'ayant pas un Rigidbody dynamic.
- Un Rigidbody dynamic, celui que l'on utilisera pour le personnage principal, permet au gameobject d'être affecté par une physique contenant la gravité, des collisions, des frictions et des forces.

Le Sprite Renderer est une composante utilisée pour afficher des images, ou sprites, sur les GameObjects dans le jeu. Il permet aux développeurs de donner une apparence visuelle aux objets en plaçant des sprites sur leur surface, ce qui est particulièrement utile pour créer des jeux en 2D.

Enfin, l'Animator est une composante permettant de créer des animations et de contrôler le mouvement des GameObjects dans le jeu. En attachant un Animator à un objet, les

développeurs peuvent définir et gérer des séquences d'animations, des transitions entre celles-ci, et contrôler le déroulement des animations en fonction des événements du jeu. Des modifications sur les différents paramètres des composants d'un gameobject peuvent être modifiés nous utiliserons cette propriété plus tard notamment pour effectuer l'attaque à l'épée

Scripting

j'ai décidé de gérer les déplacements et les attaques dans un seul script que j'ai nommé movementController.

Nous allons commencer par les déplacements basiques un par un:

Déplacements horizontaux:

Pour les déplacements nous allons utiliser des Vector2 c'est une classe de Unity qui permet de représenter un déplacement ou un emplacement et qui possède deux valeurs importantes : x et y

qui sont respectivement les composantes horizontales et verticales du déplacement ou de l'emplacement voulu. Des méthodes existent pour les Vector2 et permettent d'effectuer des opérations sur les vector2 nous en parlerons quand cela sera nécessaire.

différentes méthodes permettent de détecter l'appui sur une touche il y a:

- Input.GetMouseButton(int button) qui renvoie vrai si le bouton de la souris correspondant à l'entier passé en paramètres est appuyé faux sinon(0 clique gauche,1 clique droit,2 clique molette)
- Input.GetButton(string buttonName) qui renvoie vrai si le bouton dont le nom est entré en paramètre est appuyé faux sinon.

Pour ces deux cas si on ajoute Down ou Up après button cela détectera respectivement quand la touche vient d'être enfoncé et quand la touche est relâchée.

- Input.GetAxis(string axisName) renvoie un float compris entre -1 et 1 qui est la valeur de l'axe créé sur unity correspondant à axisName nous l'utiliserons pour l'axe horizontal et vertical qui correspondent soit aux flèches directionnels gauche et droit ou q et d (azerty) ou a et d (qwerty) pour l'horizontal et les flèches directionnels haut et bas ou z et s (azerty) ou w et s (qwerty) pour le vertical
- Input.GetAxisRaw(string axisName) renvoie -1 si Input.GetAxis renverrait un nombre négatif, 0 si il renverrait 0 ou 1 si il renverrait un nombre positif.

Je vais donc créer une variable contenant la valeur de l'axe horizontal ce qui me permettra de réutiliser sa valeur dans mon code

Et une variable contenant la vitesse du personnage que je vais pouvoir modifier dans l'inspector de unity pour cela soit on peut créer cette variable en public soit ajouter [SerializeField] avant sa déclaration, je préfère l'utilisation de [SerializeField] car la variable est en private et ne peut donc pas être accéder depuis les autres classes du projet ou de l'extérieur.

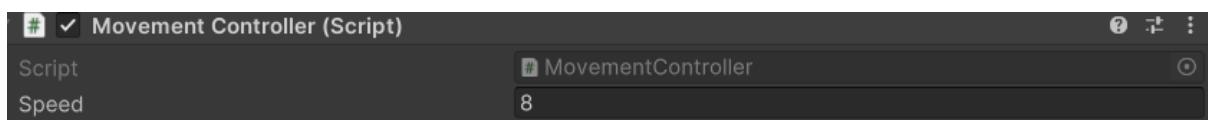
Je crée aussi un booléen canMove égal à vrai et qui vaut vrai si le personnage peut se déplacer faux sinon on changera cette valeur plus tard mais pour l'instant elle sera toujours vrai.

J'ai également ajouter [Header("Movement Settings")] qui va rajouter Movement Settings dans l'inspector pour ajouter un titre avant mes paramètres dans l'inspector et séparer les paramètres pour rendre plus simple la lecture de l'inspector.

On obtient donc la déclaration de variables suivantes.

```
[Header("Movement settings")]
private bool canMove = true;
[SerializeField] private float speed = 8f;
private float horizontal;
```

Et le résultat dans l'inspector:



Ensuite dans l'Update afin de calculer à chaque image du jeu la direction vers laquelle aller j'assigne a horizontal la valeur de Input.GetAxisRaw("Horizontal") si le joueur peut se déplacer

```
private void Update()
{
    if (canMove)
    {
        horizontal += Input.GetAxisRaw("Horizontal");
    }
}
```

Dans le fixedUpdate là où il vaut mieux effectuer les déplacements avec le rigidbody pour éviter tout problèmes de physique je change la vitesse du rigidbody attaché au joueur pour effectuer le déplacement.

le rigidbody et le collider sont stockés dans des variables au démarrage du jeu donc premièrement on les déclarent

```
... private Rigidbody2D rb;
... private Collider2D _collider;
```

puis dans Start on affecte les composants correspondant à l'aide de
getComponent<Component>()

```
70     ... private void Start()
71     {
72
73         ...     rb = GetComponent<Rigidbody2D>();
74
75         ...     _collider = GetComponent<Collider2D>();
```

Une fois que l'on a le rigidbody on a juste a changer la valeur de velocity qui est un Vector2 du rigidbody en lui affectant un nouveau vector2 avec pour x le résultat de horizontal*speed et la vitesse déjà existante sur y.

```
... Message Unity | 0 references
... private void FixedUpdate()
...
... {
...
...     if (canMove)
...
...     {
...
...         rb.velocity = new Vector2(horizontal * speed, rb.velocity.y);
...
...     }
...
... }
```

Cela permet au joueur de se déplacer de gauche à droite avec soit les flèches directionnelles, q et d (azerty) ou a et d (qwerty).

Le problème est que lorsque l'on se déplace vers la gauche le personnage continue à regarder vers la droite pour régler ça on va utiliser une fonction et modifier la valeur x du scale de transform en mettant au négatif sa valeur cela retournera le sprite du personnage et on stockera également dans une variable booléenne si le personnage est tourner vers la droite on obtient donc la déclaration suivante:

```
... private float horizontal;
... [HideInInspector] public bool isFacingRight = true;
```

[HideInInspector] permet comme son nom l'indique de cacher une variable public dans l'inspector car on ne compte pas changer sa valeur depuis l'inspector mais on veut y accéder depuis d'autres classes donc on la met en public.

On crée ensuite la fonction flip qui permet de retourner le sprite en changeant le x de localScale en son inverse elle change aussi la valeur de isFacingRight.

```
.../*
 * Change le coté vers lequel le sprite du joueur est tourné
 */
1 référence
private void Flip()
{
    if(isFacingRight && horizontal < 0f || !isFacingRight && horizontal>0f)
    {
        isFacingRight = !isFacingRight;
        Vector3 localScale = transform.localScale;
        localScale.x*=-1f;
        transform.localScale = localScale;
    }
}
```

On appellera cette fonction dans update

```
Message Unity | 0 références
private void Update()
{
}

if (canMove)
{
    horizontal = Input.GetAxisRaw("Horizontal");
    Flip();
}
```



Le sprite se retrouve bien retourné vers la gauche lors d'un déplacement vers la gauche.

Saut:

Ensuite nous allons gérer le saut pour cela nous aurons besoin d'un float correspondant à la force du saut et d'un booléen qui vaut vrai si on peut sauter.

```

    [Header("jump settings")]
    [SerializeField] private float jumpPower = 16f;
    private bool canJump = true;

```

On va créer une fonction afin de gérer le saut qui lors d'un appui sur la barre d'espace change la velocity en y du rigidbody par jumpPower et quand le personnage est en train de se déplacer vers le haut et que l'on relâche la barre espace la vitesse en y est réduite afin de retomber plus vite.

```

1 référence
private void HandleJump()
{
    if (canJump)
    {
        if (Input.GetButtonDown("Jump") && isGrounded)
        {
            rb.velocity = new Vector2(rb.velocity.x, jumpPower);
        }

        if (Input.GetButtonUp("Jump") && rb.velocity.y > 0f)
        {
            rb.velocity = new Vector2(rb.velocity.x, rb.velocity.y * 0.5f);
        }
    }
}

```

isGrounded est un booléen qui est vrai si le joueur est au sol
on déclare aussi un layermask contenant la layer sur laquelle le sol est

```

[Header("Grounded settings")]
[SerializeField] private LayerMask groundLayer;
private bool isGrounded;

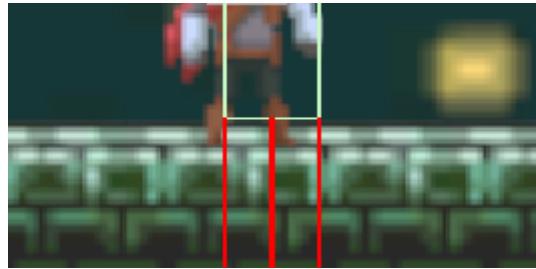
```

la valeur de isGrounded est changée au départ de l'Update par la valeur résultat de la fonction suivante:

```

private bool IsGrounded()
{
    //crée trois sorte de lasers qui détectent si le sol se trouve sur le trajet du laser renvoie vraie si sur l'un des laser le sol est trouvé faux sinon
    //bas gauche
    if ((Physics2D.Raycast(new Vector2(_collider.bounds.min.x, _collider.bounds.min.y), Vector2.down, 0.15f, groundLayer))/*gauche*/ || 
        (Physics2D.Raycast(new Vector2(_collider.bounds.center.x, _collider.bounds.min.y), Vector2.down, 0.15f, groundLayer))/*milieu*/ || 
        (Physics2D.Raycast(new Vector2(_collider.bounds.max.x, _collider.bounds.min.y), Vector2.down, 0.15f, groundLayer))/*droite*/)
    {
        return true;
    }
    return false;
}

```



Elle permet de créer trois petits rayons depuis les bordures du collider2D et allant vers le bas et si ce rayon touche un gameobject se trouvant sur la layer groundlayer la méthode renvoie vrai

On crée trois rayons, un en bas à gauche en en bas au centre et un en bas à droite pour détecter les plateformes même sur les rebords d'une plateforme. On ne peut pas utiliser Physics.CircleCast qui crée un cercle au lieu d'un rayon car il ne détecte pas le sol à cause du type de collider utilisé pour le sol

On l'appellera également dans l'Update:

```
private void Update()
{
    isGrounded = IsGrounded();

    if (canMove)
    {
        horizontal = Input.GetAxisRaw("Horizontal");
        Flip();
        HandleJump();
    }
}
```

Le saut fonctionne bien mais la chute après le saut est beaucoup trop longue on va donc modifier certaines valeurs pendant la chute et ajouter la possibilité de chuter plus rapidement.

Pour commencer on va définir les variables on aura besoin de

- un float, multiplicateur de gravité lors de la chute
- un float, multiplicateur de gravité lors de la chute rapide
- un float, gravité originale
- un float, gravité lors d'une chute
- un float, gravité lors d'une chute rapide

- Un booléen si on descend rapidement avec s ou la flèche directionnelle bas.

```
.... [Header("fall_settings")]
.... [SerializeField] float fallGravityMultiplier = 2;
.... [SerializeField] float fastFallGravityMultiplier = 3;
.... private float originalGravity;
.... private float fallGravity;
.... private float fastFallGravity;
.... private bool isCrouched = false;
```

j'ai appelé le booléen isCrouched car je ne veux pas que le joueur puisse sauter en chute rapide donc je considère le personnage comme accroupi

Dans le start j'assigne les valeurs aux différentes gravités

```
.... // Message Unity pour références
.... private void Start()
.... {
....     rb = GetComponent<Rigidbody2D>();
....     originalGravity = rb.gravityScale;
....     fallGravity = rb.gravityScale * fallGravityMultiplier;
....     fastFallGravity = rb.gravityScale * fastFallGravityMultiplier;
```

Je crée ensuite la fonction pour gérer la chute qui change les valeurs aux valeurs correspondantes dans les différents cas et j'appelle cette fonction dans update.

```

    1 reference
private void HandleFall()
{
    ...
    if (Input.GetAxis("Vertical") < 0)
    {
        canJump = false;
        isCrouched = true;
        if (rb.velocity.y > 0f)
        {
            rb.velocity = new Vector2(rb.velocity.x, 0f);
        }
        rb.gravityScale = fastFallGravity;
    }
    else
    {
        canJump = true;
        if (rb.velocity.y < 0f)
        {
            rb.gravityScale = fallGravity;
        }
        else
        {
            rb.gravityScale = originalGravity;
        }
        isCrouched = false;
    }
}

```

roulade/dash:

Pour la roulade on va juste ajouter une force dans la direction vers laquelle on est dirigé, rendre le joueur invincible puis mettre un temps avant de pouvoir réutiliser le dash on aura besoin des déclarations suivantes:

```

[Header("dash settings")]
[SerializeField] private float dashingPower = 24f;
[SerializeField] private float dashingTime = 0.2f;
[SerializeField] private float dashingCooldown = 1f;
[SerializeField] private Collider2D dashCollider;
private bool canDash = true;
private bool isDashing;

[Header("other settings")]
[SerializeField] private string ennemyLayer;
private damageHandler dmghandler;

```

damageHandler est un script que j'ai créé et que l'on parlera par la suite.

Au départ pendant le dash il pouvait arriver que l'on traverse les murs car le rigidbody ne ralentissait pas suffisamment le joueur donc en ajoutant un deuxième collider au joueur un peu plus grand que celui de base et qui est activé seulement pendant un dash il permet d'arrêter le joueur avant que l'autre collider traverse le mur.

On va donc en cas d'appui sur la touche majuscule déclencher une coroutine (ou thread) afin d'effectuer le dash et d'attendre les temps nécessaires sans arrêter complètement le jeu.

Pour se faire on utilisera le type Ienumerator une fonction de ce type doit effectuer des yield return et non des return normaux afin de pouvoir attendre un certain temps.

On obtient la coroutine suivante:

```
1 référence
··: IEnumerator Dash()
{
    ··: canDash = false;
    ··: isDashing = true;
    ··: dashCollider.enabled = true;
    ··: dmghandler.canBeDamaged = false;
    ··: Physics2D.IgnoreLayerCollision(gameObject.layer, LayerMask.NameToLayer(ennemyLayer), true);
    ··: float originalGravity = rb.gravityScale;
    ··: rb.gravityScale = originalGravity * 5f;
    ··: rb.velocity = new Vector2(transform.localScale.x * dashingPower, 0f);
    ··: yield return new WaitForSeconds(dashingTime);
    ··: dmghandler.canBeDamaged = true;
    ··: rb.gravityScale = originalGravity;
    ··: isDashing = false;
    ··: dashCollider.enabled = false;
    ··: Physics2D.IgnoreLayerCollision(gameObject.layer, LayerMask.NameToLayer(ennemyLayer), false);
    ··: yield return new WaitForSeconds(dashingCooldown);
    ··: canDash = true;
}
```

Que l'on appellera à l'aide de la méthode StartCoroutine(IEnumerator routine)

```
1 référence
··· private void HandleDash()
··· {
··· ··· if (Input.GetKeyDown(KeyCode.LeftShift) && canDash && isGrounded)
··· ··· {
··· ··· ··· StartCoroutine(Dash());
··· ··· }
··· }
```

On appellera cette fonction dans l'update.

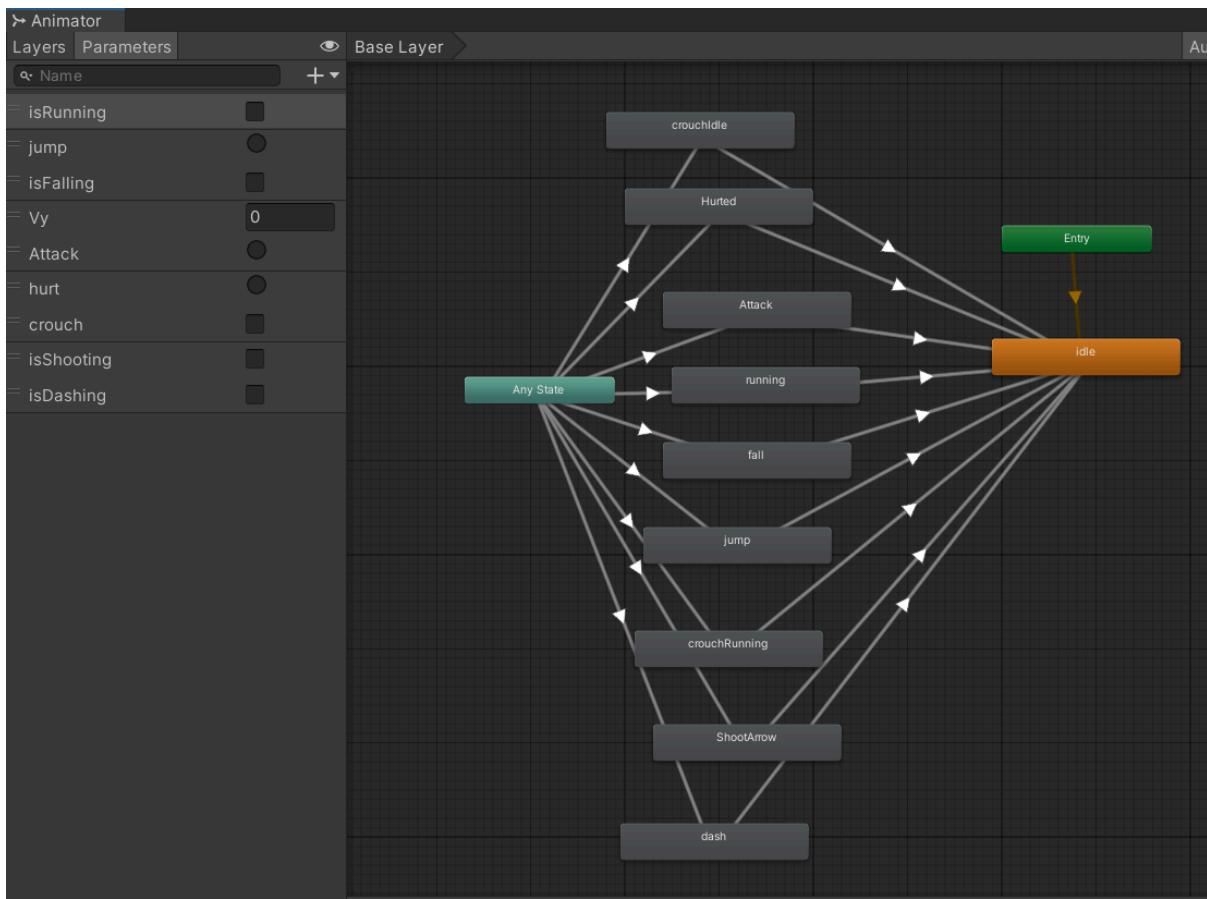
Pendant le dash, le joueur n'a plus de contrôle sur le personnage, on peut l'implémenter simplement en ajoutant au début du Update et du FixedUpdate :

```
··· if (isDashing)
··· {
··· ··· return;
··· }
```

ce qui va quitter la fonction directement sans effectuer de mouvements

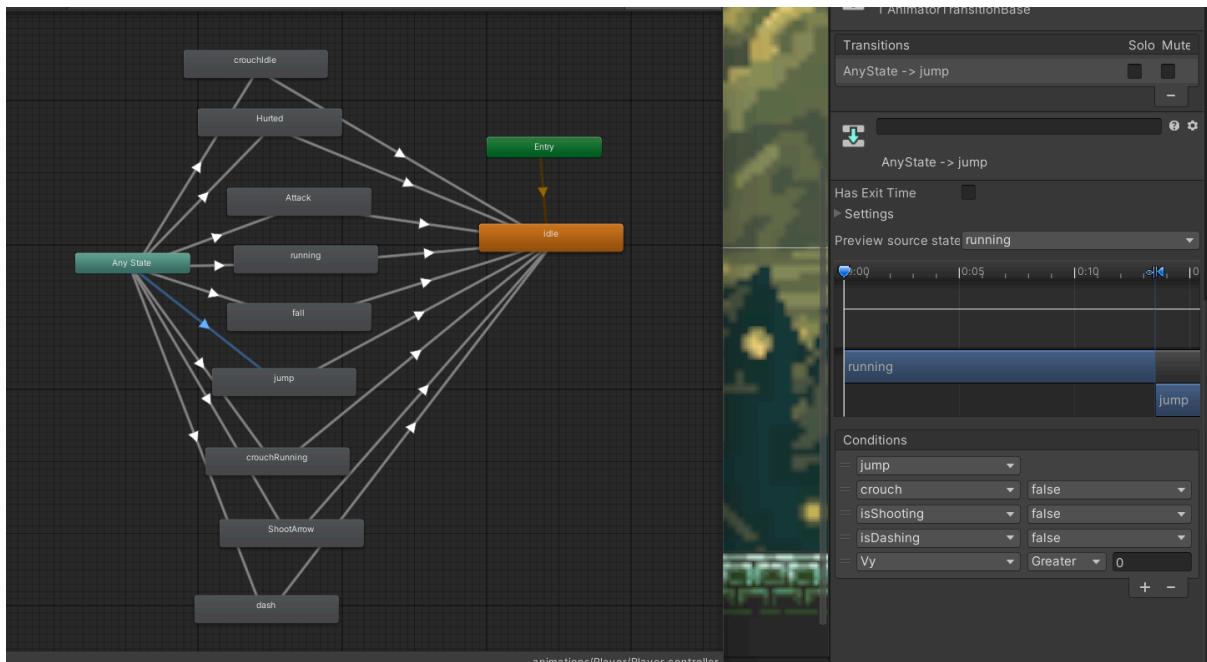
Animations:

Pour les animations il faut d'abord comprendre comment l'Animator fonctionne:



A gauche vous avez une liste de variables et à droite des états qui sont des animations avec différentes transitions.

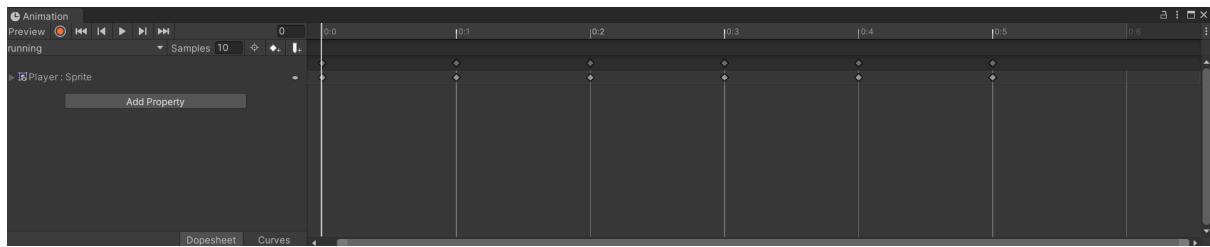
Quand on clique sur une transition elle s'ouvre dans l'inspector et on peut modifier différents paramètres



has exit time veut dire qu'il attendra la fin de l'animation précédente afin de lancer la suivante

On peut changer le temps qu'il faut pour passer d'une animation à l'autre avec le graphique et enfin en bas les variables avec leurs valeurs nécessaires afin que l'animation soit effectuée.

Il y a différents types de variables, ceux classiques comme float ou boolean et d'autres comme le trigger un trigger ressemble beaucoup à un booléen mais sera remis à faux après la transition.



Les animations sont modifiées grâce à cette fenêtre elle permet de modifier les valeurs des composants d'un gameobject le plus souvent elles sont utilisées pour modifier le sprite du sprite renderer afin de changer l'image utilisée pour le gameobject et donner du mouvement à l'image. Mais une animation peut aussi être utilisée pour désactiver un gameobject ou changer sa vitesse on utilisera cette propriété pour effectuer l'attaque à l'épée.

Revenons aux variables, les valeurs des variables de l'animator peuvent être changés depuis un script en utilisant les méthodes suivantes par exemple:

- animator.SetBool(string nomDeLaVariable, bool valeur)
- animator.SetFloat(string nomDeLaVariable, float valeur)
- animator.SetTrigger(string nomDuTrigger)

On peut également forcer le trigger a se remettre à faux avec
animator.ResetTrigger(string nomDuTrigger)

regardons notre script:

premièrement on va déclarer et assigner l'animator

```
... Animator animator;
    ↵ Message Unity | 0 références
... private void Start()
...
...
rb = GetComponent<Rigidbody2D>();
originalGravity = rb.gravityScale;
fallGravity = rb.gravityScale * fallGravityMultiplier;

fastFallGravity = rb.gravityScale * fastFallGravityMultiplier;
animator= GetComponent<Animator>();
    ↵     ...
```

ensuite dans les fonction où c'est nécessaire on va modifier les valeurs de l'animator , la plus importante modification c'est l'ajout de la fonction animate qui anime certains mouvements, elle montre bien les modifications des

valeurs de l'animator:

```
private void Animate()
{
    if (!isDashing)
    {
        animator.SetBool("crouch", isCrouched);
        if (isGrounded)
        {

            animator.SetBool("isFalling", false);
            if (horizontal != 0f)
            {
                animator.SetBool("isRunning", true);
                idleCheck = 5;
            }
            else
            {
                if (idleCheck > 0)
                {
                    idleCheck--;
                }
                else
                {
                    animator.SetBool("isRunning", false);
                }
            }
        }
        else
        {
            animator.SetBool("isFalling", true);
        }
    }
}
```

Attaques:

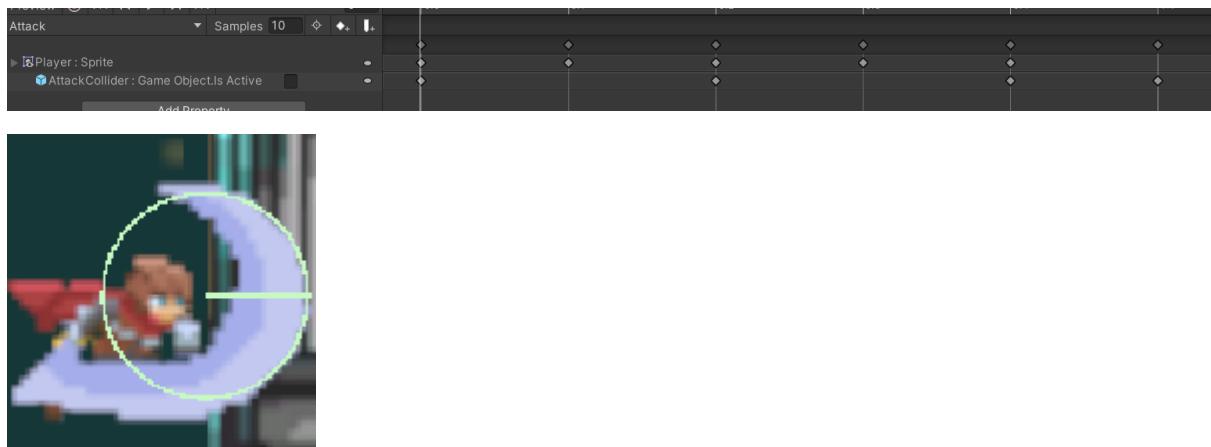
On a décidé d'avoir la possibilité de faire deux attaques différentes une à l'épée et une à l'arc le lancement de l'attaque est gérée par une seule fonction:

```

private void HandleAttack()
{
    if (Input.GetMouseButtonDown(0) && !isShootingArrow)
    {
        animator.SetTrigger("Attack");
    }
    if (Input.GetMouseButtonDown(1) && canShootArrow)
    {
        StartCoroutine(Arrow());
    }
}

```

pour l'attaque à l'épée lors de l'appui sur le clic gauche de la souris on change le trigger de l'Animator Attack qui nous fait passer à l'animation suivante:

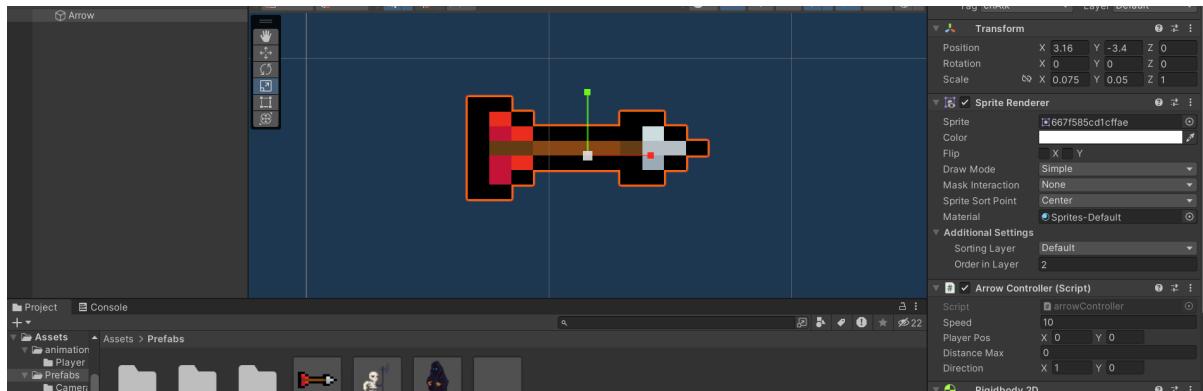


A la fin de l'animation quand l'attaque est censée faire des dégâts le gameobject Attack collider devient actif ce qui permet d'effectuer des dégâts aux ennemis se trouvant dans cette zone

pour l'attaque à l'arc on va utiliser un prefab

Les prefabs abréviation de préfabriqués sont des gameobject sauvegardés et qui permettent de ne pas recréer le gameObject depuis zéro à chaque fois, ils se modifient de la même façon qu'un gameObject normal et les valeurs de ses composants peuvent être également modifiées.

Voici le prefab de la flèche



Il ne reste plus qu'à instancier ce prefab lors du lancer de la flèche à l'aide de la méthode instantiate

```
GameObject arrow = Instantiate(arrowPrefab, new Vector2(transform.position.x, transform.position.y), isFacingRight ? Quaternion.identity : Quaternion.Euler(0f, 0f, 180f));
arrowController ac = arrow.GetComponent<ArrowController>();
ac.direction = Vector2.right;
ac.playerPos = transform.position;
ac.distanceMax = arrowMaxDistance;
```

Les paramètres sont le prefab, la position où l'instancier et sa rotation qui est représenté à l'aide de quaternions.

l'instantiate est fait à la fin de l'animation du tir.

puis les valeurs de l'arrow controller sont modifiés

l'arrowController est un script qui fait se déplacer la flèche et en cas de contact avec un ennemi ou un mur ou quand elle s'éloigne de trop elle est détruite. Si elle touche un ennemi, elle lui inflige des dégâts.

points de vie, dégâts et game over

Pour les points de vie et les dégâts on commence par tout définir

```
[Header("health settings")]
[SerializeField] private int maxHealth = 20;
5 références
public int currentHealth { get; private set; }

[Header("on damage settings")]
public bool canBeDamaged = true;
[SerializeField] private int staggerTime;

[Header("damaging tags settings")]
[SerializeField] private string ennemyTag;
[SerializeField] private string ennemyAttackTag;
[SerializeField] private HealthPointUIController hpController;
MovementController moveControl;
// Script à venir pour faire le dégât
```

Ici le getter de currentHealth est en public mais son setter est en privé cela permet de pouvoir récupérer sa valeur sans pour autant pouvoir la modifier depuis une autre classe.

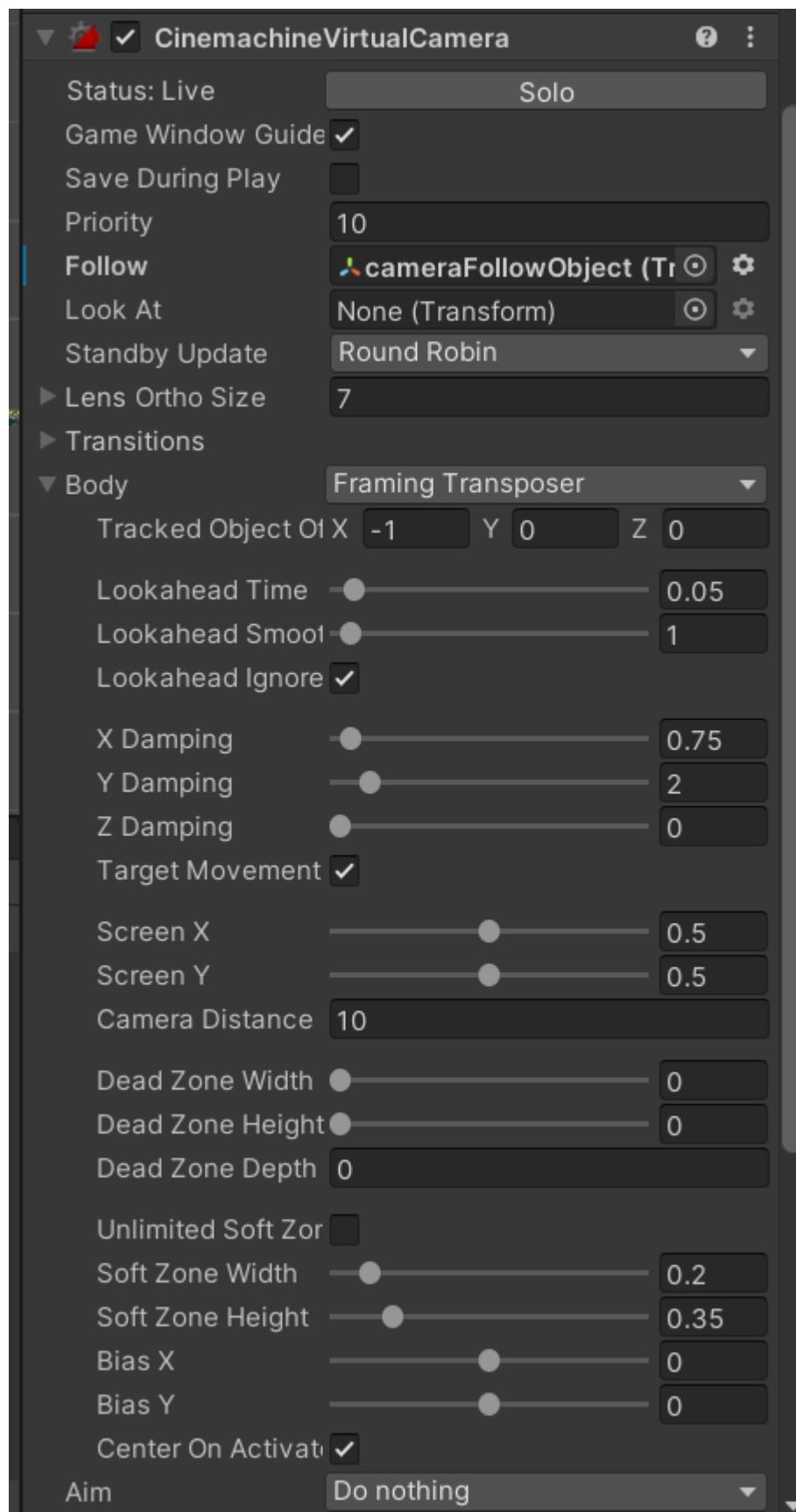
On va utiliser l'événement OnCollisionEnter2D afin de détecter les collisions avec les ennemis

```
Message Unity | 0 références
private void OnCollisionEnter2D(Collision2D collision)
{
    if (canBeDamaged && !collision.collider.isTrigger && (collision.gameObject.CompareTag(ennemyTag) || collision.gameObject.CompareTag(ennemyAttackTag)))
    {
        currentHealth--;
        hpController.damage(currentHealth);
        if (currentHealth <= 0)
            death();
        else {
            if (collision.gameObject.CompareTag(ennemyTag))
                stagger(collision.gameObject.transform.position);
            if (collision.gameObject.CompareTag(ennemyAttackTag))
                stagger(collision.gameObject.GetComponentInParent<Transform>().position);
        }
    }
}
```

et dans le cas où on touche un ennemi on va prendre un dégat et le personnage va être repoussé et incapable de bouger ou attaquer pendant qu'il se fait repoussé pour ça dans le movement controller on changera la valeur de canMove à false.

Développement Camera:

On souhaite avoir une caméra qui suivra le joueur avec un délai qui n'est donc pas directement accrochée au joueur pour cela, Unity possède un package nommé cinemachine qui permet de modifier comment la caméra se comporte, ici nous n'allons que utiliser les possibilités de déplacements du cinemachine. Les valeurs choisies sont celles qui permettent que la caméra s'approche le plus possible de ce que l'on souhaite.



On a également ajouté un gameobject nommé cameraFollowObject qui suit le joueur et quand le joueur change de direction ce gameobject change de côté par rapport au joueur de façon lisse et agréable à l'aide d'une coroutine utilisant la méthode Mathf.Lerp(valeurDepart,valeurArivee,t) qui renvoie un float qui est le résultat d'une interpolation entre la valeur de départ et celle d'arrivée par rapport à t et pour calculer t on prend le temps qu'il s'est écoulé depuis le début de la coroutine que l'on divise par le temps total que doit faire le déplacement on a donc le code suivant:

```

1 référence
private IEnumerator FlipYLerp()
{
    float startRotation=transform.localEulerAngles.y;
    float endRotationAmount = DetermineEndRotation();
    float yRotation = 0f;

    float elapsedTime = 0f;
    while(elapsedTime < flipYRotationTime)
    {
        elapsedTime +=Time.deltaTime;

        yRotation=Mathf.Lerp(startRotation, endRotationAmount, elapsedTime/flipYRotationTime);
        transform.rotation = Quaternion.Euler(0f, yRotation, 0f);

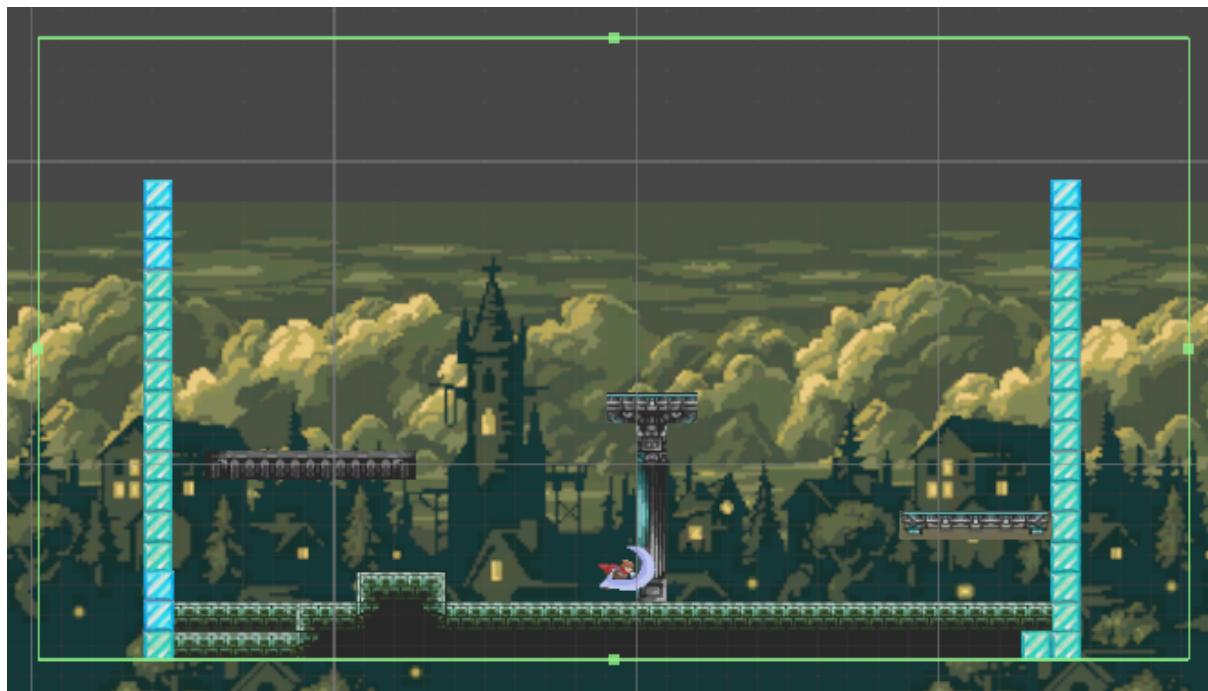
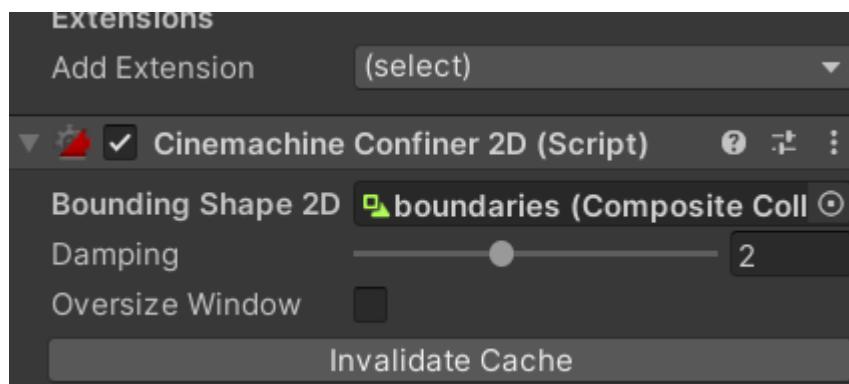
        yield return null;
    }
}

1 référence
private float DetermineEndRotation()
{
    isFacingRight = !isFacingRight;
    if (isFacingRight)
    {
        return 180f;
    }
    return 0f;
}

```

On ajoute également un script qui va réduire la vitesse de suivi quand on saute puis l'augmenter quand on retombe.

la caméra suit bien le personnage comme on le souhaite mais elle descend trop sous la carte et contre les murs pour régler cela, dans cinemachine on a une extension du cinemachine nommé cinemachine boundaries auquel on peut passer un gameobject possédant un composite collider qui est un assemblage de différents collider et qui prend ces limites pour bloquer la caméra.



On a un personnage capable de se déplacer, d'attaquer avec un système de vie. Et une caméra qui suit le joueur proprement.

Presque tous les éléments prévus au départ ont pu être mis en place, Mais l'ajout d'une façon de récupérer de la vie et des petits détails sur les sauts et l'Animator auraient pu être améliorés.

Développement Enemies:

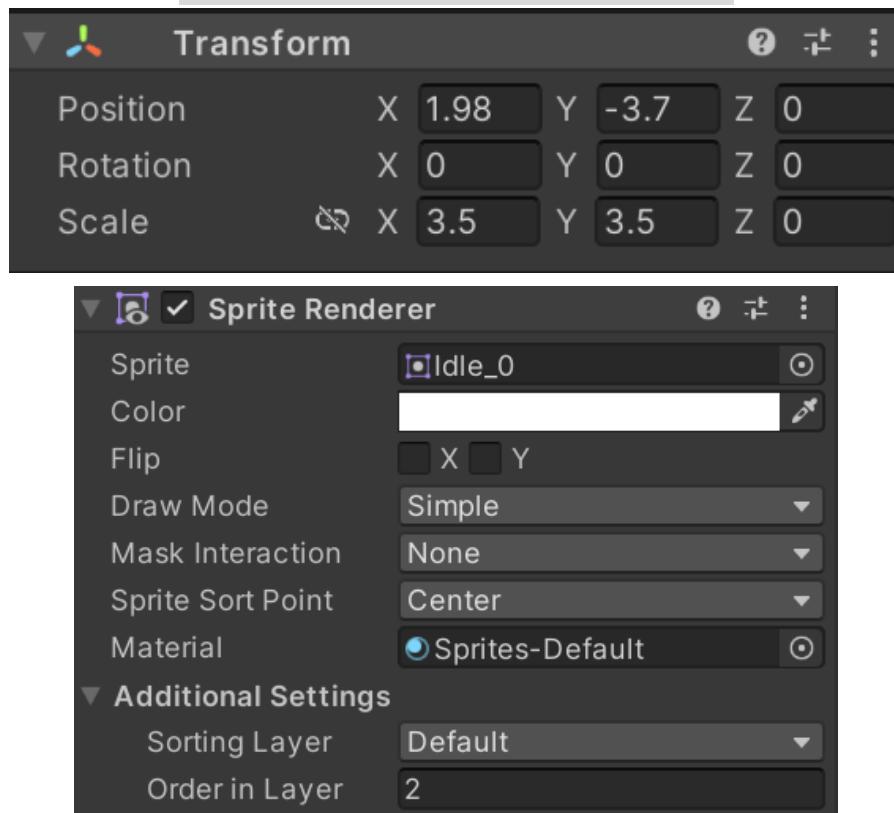
Avant de commencer les scripts , en unity on va toujours devoir créer le GameObject et ses composantes. Pour choisir les composantes nécessaires au GameObject on s'est posé la question, déjà comment les physiques de notre jeux vont être gérés. Unity comme dit précédemment nous propose dedans ces composantes qui viennent avec beaucoup des propriétés et qui vont nous faciliter la gestion graphique ,physique ,audiovisuelle entre autres.

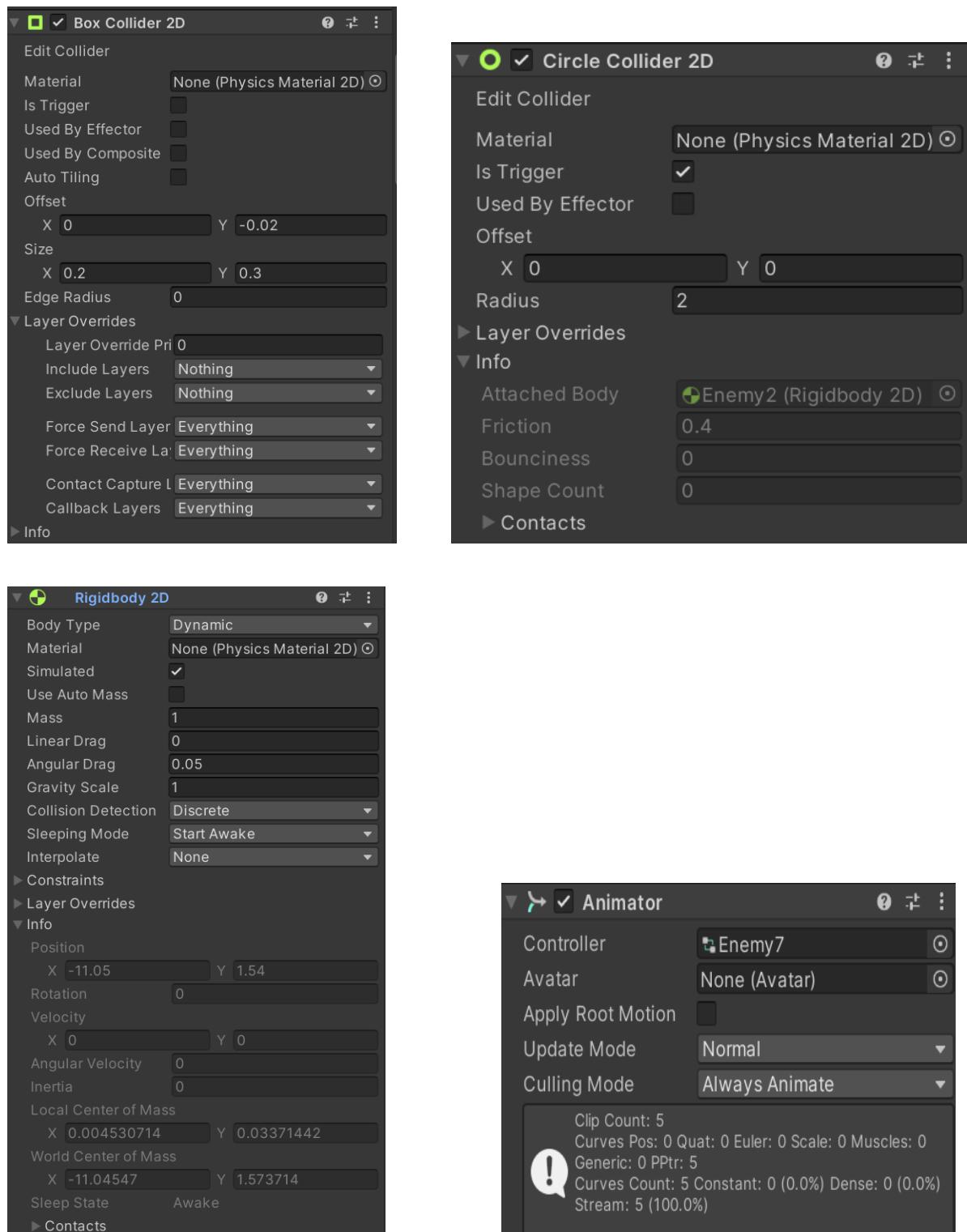
Composition du GameObject

Les composantes qui nous vont être utiles sont :

- Transform
- Collider2d (circle, Box)
- RigidBody
- Sprite Renderer
- Animator

Voici en Unity dans la fenêtre “Inspector”:





Comme référencé précédemment , dans Unity, les GameObjects sont enrichis par diverses composantes qui définissent leur comportement, leur apparence et leurs interactions dans le jeu. Comme on le voit chaque composante est composée des plusieurs paramètres que l' on peut facilement manipuler pour mettre en place le GameObject qu'on veut.

Transform: Le Transform, détermine la position, la rotation par des vecteur et l'échelle d'un GameObject dans l'espace tridimensionnel du jeu. Grâce au Transform, les développeurs peuvent placer et orienter les objets de manière précise dans la scène.

Colliders2D: Pour gérer les collisions et les interactions physiques dans le jeu, Unity offre plusieurs types de colliders, tels que CircleCollider2D et BoxCollider2D. Ces composantes permettent de définir des zones de collision circulaires ou rectangulaires autour des GameObjects, facilitant ainsi la détection des collisions avec d'autres objets et la gestion des interactions physiques ou artificielles.

RigidBody: Le RigidBody est une composante essentielle pour simuler les comportements physiques réalistes des objets dans le jeu. En attachant un RigidBody à un GameObject, celui-ci devient sujet aux lois de la physique, telles que la gravité et les forces externes, ce qui lui permet de se déplacer, de tomber, ou de réagir aux collisions de manière réaliste. Ceci peut ne pas sembler instinctif, étant donné qu'il s'agit d'un platformer qui, en soi, ne simule pas de physiques réelles. Cependant, l'utilisation d'un RigidBody présente des avantages, comme la gestion automatique de la gravité (une fonction en moins à faire dans le script) et la gestion des vitesses.

D'un autre côté, l'implémentation de cette composante dans notre jeu pose également certains problèmes, tels que la friction et les interactions entre les corps des ennemis qui peuvent être poussés par d'autres corps et perturber leurs actions automatiques. La friction pose problème car si des forces sont utilisées pour manipuler les ennemis, il est possible qu'ils restent coincés dans un mur pendant qu'il est en l'air en raison d'une force constante qui les maintient appuyés contre le mur, alors qu'on voudrait qu'ils glissent et tombent au sol.

Le paramètre dans la composante que Unity nous offre pour résoudre ce type de problème est l'option du type de RigidBody, kinematic ou dynamic. Lorsqu'un Rigidbody est configuré en tant que dynamic, il est, comme mentionné précédemment, soumis aux lois de la physique, ce qui signifie qu'il peut avoir des mouvements résultant de ces interactions physiques. Mais lorsqu'un Rigidbody est défini en tant que kinematic, il est contrôlé manuellement par le code du jeu plutôt que par les lois de la physique. Ainsi, à certains

moments, nous pouvons changer par code le type de corps d'un personnage de dynamique à kinematic en fonction de nos besoins.

Sprite Renderer: Le Sprite Renderer est une composante utilisée pour afficher des images, ou sprites, sur les GameObjects dans le jeu. Il permet de donner une apparence visuelle aux objets en plaçant des sprites sur leur surface, ce qui est particulièrement utile pour créer des jeux en 2D.

Animator: l'Animator est une composante permettant de créer des animations et de contrôler le mouvement des GameObjects dans le jeu. En attachant un Animator à un objet, les développeurs peuvent définir et gérer des séquences d'animations, des transitions entre celles-ci, et contrôler le déroulement des animations en fonction des événements du jeu.

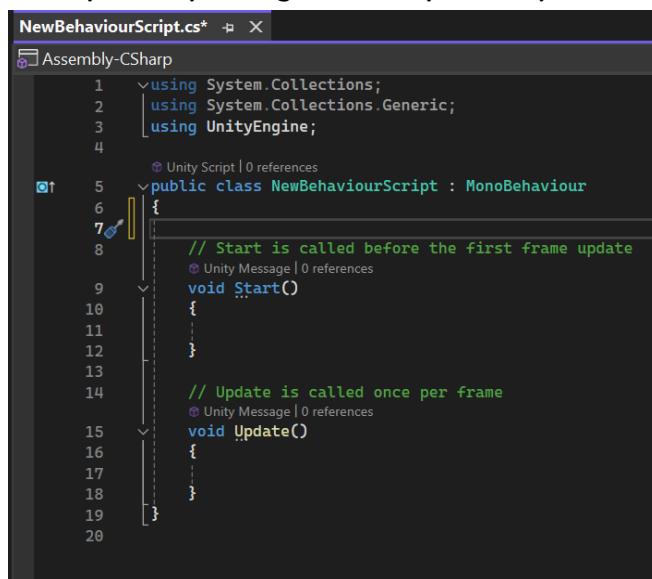
En résumé, les composantes telles que Transform, Colliders 2D, Rigidbody, Sprite Renderer et Animator vont m'offrir les outils nécessaires pour positionner, animer, afficher et simuler le comportement des ennemis dans le jeu.

Une autre petit détail qui faudra prendre en compte, est l'utilisation des Tags et Layers

Scripting

Le scripting sur Unity avec C# est au cœur du processus de développement du jeu. Au centre de cette approche se trouve la classe abstraite MonoBehaviour, qui est la base de tous les scripts dans Unity. Cette classe fournit des méthodes prédéfinies telles que Update, FixedUpdate, Awake et Start, qui sont essentielles et nécessaires pour contrôler le comportement des GameObjects pendant le déroulement du jeu.

Exemple Script vierge donnée par Unity:



```
NewBehaviourScript.cs* ✎ X
Assembly-CSharp
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class NewBehaviourScript : MonoBehaviour
6  {
7
8      // Start is called before the first frame update
9      void Start()
10     {
11
12     }
13
14      // Update is called once per frame
15      void Update()
16     {
17
18     }
19
20 }
```

La fonction Update est appelée à chaque frame (image) du jeu et est utilisée pour mettre à jour le comportement des GameObjects en fonction du temps écoulé depuis le dernier frame. Cela permet de gérer les interactions en temps réel, telles que les mouvements du joueur, les entrées utilisateur et les animations.

D'autre part, la fonction FixedUpdate qui se déroule avant Update est utilisée pour les calculs physiques et est appelée à intervalles de temps fixes, indépendamment de la vitesse du jeu. Cela garantit des simulations physiques stables et cohérentes, notamment pour les mouvements et les interactions basées sur la physique.

La fonction Awake est appelée lorsque le script est initialisé, avant le démarrage du jeu. Elle est principalement utilisée pour initialiser les variables et les objets nécessaires au fonctionnement du script. Quant à la fonction Start, elle est appelée une seule fois au début du jeu, juste avant que la première frame ne soit rendue. C'est l'endroit idéal pour effectuer des initialisations qui nécessitent que d'autres objets soient déjà initialisés, comme la recherche d'autres GameObjects dans la scène.

Ces fonctions sont mises en relation de manière cohérente par Unity pour assurer le bon fonctionnement du jeu. L'ordre d'exécution est généralement le suivant : Awake est appelé en premier, suivi de Start, puis FixedUpdate et Update sont appelées de manière continue pendant le déroulement du jeu. Cette séquence garantit que les scripts sont initialisés correctement et que les comportements sont mis à jour de manière appropriée dans chaque frame du jeu, assurant ainsi une expérience de jeu fluide et cohérente.

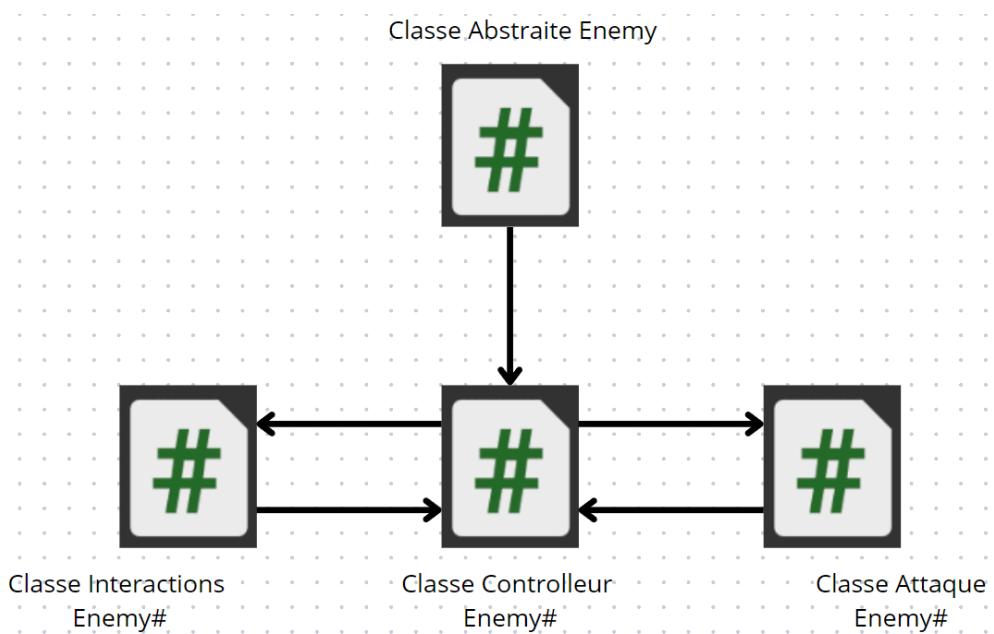
Code des Ennemis

Dans cette partie, je vais passer en revue le code de chaque ennemi, expliquant ses fonctions principales, ses caractéristiques particulières, ce que j'ai dû apprendre, les détails auxquels j'ai dû prêter attention, et comment j'ai résolu ces problèmes.

À part la classe parent déjà annoncée dans la conception, que nous remplirons au fur et à mesure. En général, j'ai divisé le code en 3 scripts qui communiquent entre eux pour mieux organiser la programmation. En effet, même si les ordinateurs et Unity sont capables de lire des millions de lignes de code, trop de code peut nous faire perdre facilement le fil de ce que nous sommes en train de faire, rendant le développement plus lent et confus. Ainsi, ces 3 scripts seront les suivants:

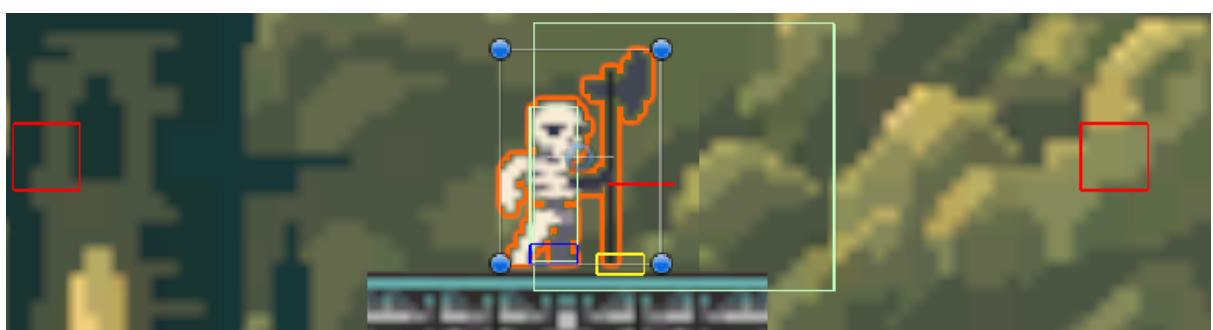
- **Le script contrôleur**, qui est le script principal, héritera de la classe Enemy et auquel les 2 autres scripts seront associés. Ce script gérera les fonctionnalités principales, ainsi que ces méthodes et attributs uniques et celles qu'il devra réimplémenter du parent.

- **Le script Interactions**, s'occupera de gérer les collisions et interactions. À l'intérieur, il contiendra les méthodes principales permettant d'identifier si l'ennemi a collisionné avec le sol, le personnage, un mur, etc. Il communiquera ensuite le résultat au script contrôleur, qui se chargera de gérer la réponse à ces collisions dans son propre script.
- **Le script Attaque**, sera attaché à un GameObject enfant de l'ennemi, qui gèrera les colliders représentant les zones d'attaque de l'ennemi. J'ajoute ce script car il sera placé dans un GameObject enfant de l'ennemi, ce qui évite d'ajouter trop de colliders dans un même GameObject. En effet, cela saturerait le GameObject et le rendrait plus difficile à gérer. Ce script communique avec le script contrôleur pour savoir quand désactiver ou activer les nouveaux colliders.



Squelette:

Pour commencer , j'ai décidé de évidemment coder l'ennemie le plus simple, comme ca je découvre le codage en c# avec les fonctions de MonoBehaviour.



Script controller:

Le mouvement

Pour commencer le script de mon premier ennemi, j'ai d'abord décidé de mettre en place son mouvement de base. Dans la phase d'initialisation du code, j'ai défini deux vecteurs 2D (héritée du père) pour délimiter la zone de patrouille du squelette. Ensuite, j'ai réimplémenté une fonction "mouvement" ainsi que "getDir", des fonctions déjà déclarées dans le parent enemy. Dans la fonction de mouvement, j'ai utilisé des attributs de type caractère également hérités du parent, tels que "currentAction" et "nextAction". Cette méthode est construite autour d'une instruction switch pour gérer les différentes valeurs possibles de "currentAction". Les actions possibles pour le squelette sont : 'W' pour Wait (attendre), qui appelle une autre méthode d'attente, 'L' pour Left (aller à gauche), 'R' pour Right (aller à droite) et 'A' pour attaquer, qui appelle une méthode d'attaque. À la fin de chaque action, que ce soit lorsque l'ennemi a fini d'attendre, d'attaquer ou lorsqu'il atteint les limites de sa zone de patrouille, il appelle la fonction "getDir". En fonction de la valeur actuelle de "currentAction", cette fonction va soit échanger sa valeur par celle de "nextAction", soit lui assigner directement une nouvelle valeur.

Pour le mouvement de l'objet, nous utiliserons les paramètres de vitesse de sa composante Rigidbody. Simplement, nous modifions la valeur de son paramètre velocity dans le script pour qu'il se déplace vers la gauche ou la droite. Il est important de noter qu'il commence son mouvement dès qu'il touche le sol (tous les ennemis apparaissent dans l'aire pour éviter qu'ils ne pénètrent dans les structures). De plus, à ce moment-là, lorsqu'il est détecté grâce au script interaction (sur lequel nous reviendrons en détail par la suite) qu'il est au contact du sol, l'objet devient kinematic.

Détails imprévu à gérer

Une fois que les mouvements de base ont été programmés , je me suis rendu compte , de certains détails, auxquels il fallait faire attention. Le première détail était, de gérer le cas où le squelet rencontre une structure en face de lui et il ne peut plus continuer son chemin, et le deuxième détail étant de gérer le cas où l'ennemi se retrouve dans un plateforme et avant d'atteindre la limite de sa zone à patrouiller il se retrouve avec la fin de la plateforme.

Pour résoudre ces contraintes, dans le script interactions j'ai découvert l'utilisation des fonctions Ray/BoxCast dedans les physiques proposés par Unity.

Après avoir créé ceux-ci dans le script Interactions pour résoudre ces détails, je crée deux méthodes. La première gère le cas où le script Interactions renvoie "vrai" lorsque l'ennemi rencontre un obstacle. Lorsqu'il reçoit ce statut, il le garde dans une variable booléenne. Si c'est vrai, il appelle la méthode de prévention qui identifie dans quelle direction l'ennemi se dirige, le redirige vers sa direction opposée et réinitialise la valeur du vecteur 2D position qui définit la limite de son parcours. En fonction de s'il est plus proche d'une limite ou de l'autre, elle change le vecteur approprié.

Ensuite, la deuxième méthode est similaire, à la différence qu'elle est appelée uniquement lorsque le script Interactions renvoie "faux" lorsque l'ennemi va tomber d'une plateforme.

Elle est appelée lorsque dans une variable booléenne la valeur est "faux" car, comme nous l'expliquerons dans la partie suivante, le script Interactions utilise un boxcast pour identifier s'il y a du sol ou si l'ennemi se dirige vers le vide.

L'attaque

Son attaque est gérée à travers une méthode qui hérite du père et est réimplémenté. Cette méthode est conçue de manière simple: pendant une durée calculée en fonction de la durée de l'animation d'attaque du squelette, la méthode modifie la valeur d'un attribut booléen appelé "isAttacking". Cette valeur indique au script d'attaque si les colliders de la zone d'attaque doivent être activés ou non. En activant ces colliders pendant la phase d'attaque et en les désactivant après un certain laps de temps déterminé par la durée de l'animation.

La mort et destruction de l'objet

Pour la gestion de sa mort et de sa destruction, chaque ennemi possède un attribut nombre entier définissant ses points de vie. Lorsque cet attribut atteint une valeur de zéro, l'ennemi est considéré comme battu. À ce moment-là, une méthode réimplémentée du parent, appelée "Dies", est invoquée. Cette méthode met en œuvre plusieurs actions cruciales : elle active un booléen pour indiquer que l'ennemi est mort, configure son collider en mode trigger pour désactiver les collisions, et met sa gravité à zéro pour éviter qu'il ne tombe du niveau, étant donné que son collider devient un trigger. Une fois que le booléen annonce que l'ennemi est mort, le script déclenche une méthode héritée du parent, "toDestroy", qui initialise un compte à rebours de 4 secondes avant de détruire définitivement l'objet ennemi. Cette approche garantit une transition fluide de la vie à la mort de l'ennemi, suivie d'une destruction contrôlée de son objet.

Script interactions:

Les raycasts et boxcasts

Les raycasts et boxcasts sont des techniques pour détecter les collisions et les intersections entre des objets dans le jeu. Les raycasts émettent un rayon invisible depuis un point donné dans une direction spécifique, tandis que les boxcasts émettent un volume rectangulaire (une boîte) pour détecter les collisions dans une direction donnée.

(Pour avoir une idée, on peut regarder dans l'image précédente, les rayons en cercle rouge et les rectangles bleus, jaunes).

Après avoir abordé l'utilisation des raycasts, des boxcasts et des méthodes pour anticiper les situations où l'ennemi pourrait rencontrer un obstacle sur son chemin ou risquerait de tomber d'un étage, je me concentre maintenant sur la création et réception des valeurs booléennes correspondantes. Ces valeurs booléennes sont essentielles pour indiquer à l'ennemi s'il peut se déplacer librement ou s'il doit prendre des mesures pour éviter les obstacles ou les chutes. En les recevant correctement et en les interpréter avec précision dans le script contrôleur, je m'assure que son comportement est réactif à son environnement.

Dans le script d'interaction, les raycasts et boxcasts sont utilisés pour renvoyer des valeurs booléennes au script controller. Trois méthodes sont mises en place à cet effet : une

utilisant un raycast et deux utilisant des boxcasts. La première méthode, qui utilise un raycast, génère une ligne rouge devant le personnage et renvoie true si cette ligne touche une structure, ce qui permet de prévenir les obstacles. Les deux autres méthodes, utilisant des boxcasts, ont des objectifs distincts : la première crée un rectangle au niveau des pieds du personnage et renvoie true si elle touche le sol, indiquant ainsi si le personnage est au sol ; la seconde crée un rectangle à la hauteur des pieds du personnage et en face de lui, renvoyant true si elle touche le sol, ce qui permet de détecter si le personnage risque de tomber des plateformes. Cette approche basée sur les raycasts et boxcasts offre une manière distincte au collider plus efficace et précise.

Méthodes OnCollision, OnTrigger

En Unity il existe des méthodes de MonoBehaviour pour gérer facilement la détection de collision et interactions entre objets, les **OnCollision** et **OnTrigger**. Plus précisément il existait 6 méthodes qu'on peut implémenter **OnCollisionEnter**, **OnCollisionStay**, **OnCollisionExit**, **OnTriggerEnter**, **OnTriggerStay** et **OnTriggerExit**. Chacune de ces méthodes a un rôle spécifique et est déclenchée dans des circonstances particulières. Les différences principales entre les méthodes **OnCollision** et **OnTrigger** résident dans le type de collision qu'elles détectent.

OnCollisionEnter est appelée lorsque deux colliders entrent en collision pour la première fois. Utile pour détecter le début d'une collision entre deux objets physiques dans le jeu.

OnCollisionStay est appelée tant que la collision entre deux colliders persiste, ce qui permet de détecter les interactions continues pendant la collision.

OnCollisionExit est appelée lorsque deux colliders cessent de se toucher, signalant ainsi la fin de la collision.

OnTriggerEnter, **OnTriggerStay** et **OnTriggerExit** sont utilisées pour détecter les collisions entre un collider et un trigger, qui est un collider qui n'a pas d'effet physique mais déclenche des événements lorsqu'un autre collider entre en contact avec lui.

OnTriggerEnter est déclenchée lorsque le collider entre dans la zone de déclenchement du trigger.

OnTriggerStay est appelée tant que le collider est à l'intérieur de la zone de déclenchement.

OnTriggerExit est déclenchée lorsque le collider sort de la zone de déclenchement du trigger.

Gestion attaque et détection joueur

Pour gérer les collisions liées à l'attaque du joueur et à l'entrée du joueur dans la zone de vue de l'ennemi, je vais utiliser les méthodes **OnCollisionEnter** et **OnTriggerEnter***. Tout d'abord, je vais utiliser **OnCollisionEnter** pour détecter l'attaque du joueur. Lorsque le joueur

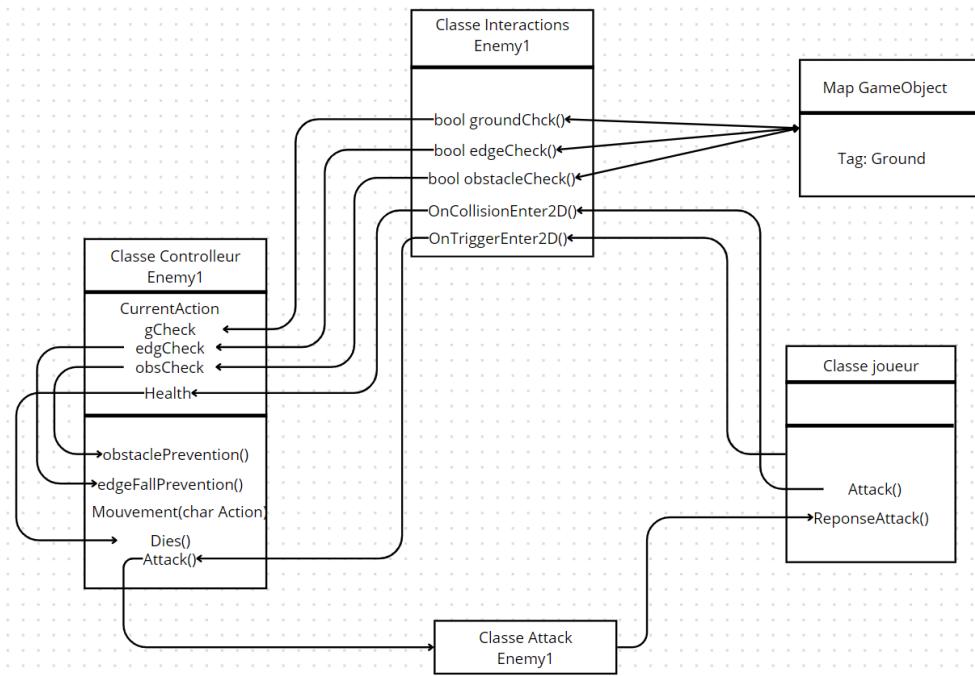
effectue une attaque et que son collider entre en collision avec celui de l'ennemi, la méthode OnCollisionEnter sera déclenchée, ce qui déclenche une réponse appropriée dans le script Interaction l'ennemi qui communiquera au script controller pour gérer les dégâts subis ou d'autres actions liées à la réponse de l'attaque.

Ensuite, j'utiliserai OnTriggerEnter pour détecter quand le joueur entre dans la zone de vision de l'ennemi et déclencher l'attaque de l'ennemi. En plaçant un trigger collider plus grand que la zone d'attaque je pourrai détecter lorsque le joueur pénètre dans cette zone et s'approche du squelette. Cela déclenche des actions dans le script de l'ennemi, telles que l'activation de l'attaque du joueur en changeant la variable "currentAction". En combinant ces deux méthodes de collision, je crée des interactions dynamiques et réactives entre le joueur et les ennemis dans le jeu.

Attack script:

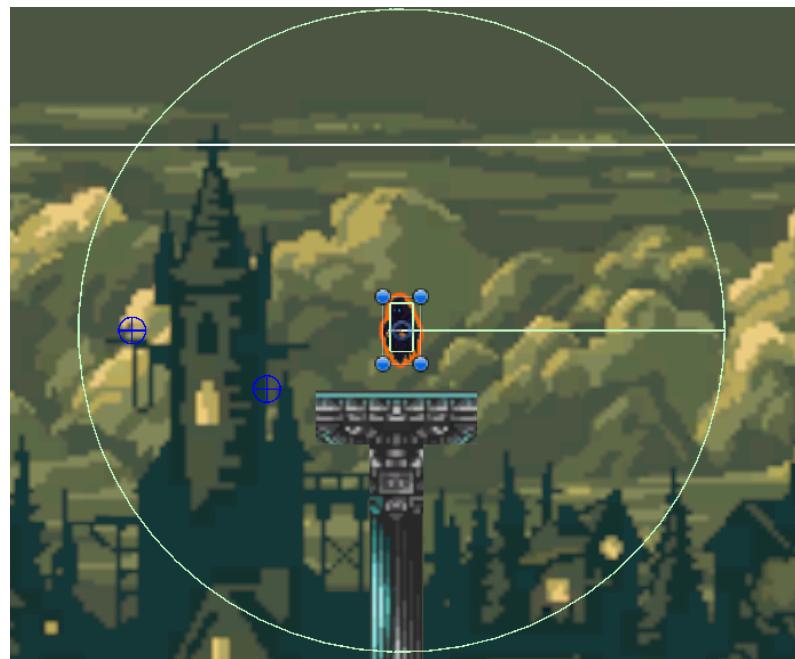
Pour organiser efficacement le code, j'ai créé un troisième petit script. Certaines entités ennemis, comme le squelette, nécessitent l'utilisation de hit boxes pour définir leur méthode d'attaque. Pour ce faire, j'ai ajouté un GameObject enfant, contenant des colliders représentant ces zones d'attaque. Cette approche permet d'éviter d'ajouter trop de colliders au GameObject principal de l'ennemi, ce qui risquerait de compliquer sa gestion. En regroupant les hitboxes dans un GameObject fils distinct, il devient plus facile de les organiser et de les gérer, tout en améliorant la lisibilité et la maintenabilité du code.

Diagramme de classes



Fantôme:

Maintenant qu'on aborde les découvertes les plus importante en codant pour le première ennemi et pour la première fois en c# on va omettre beaucoup des détails et méthodes qui vont se répéter dans les script des prochain ennemis. On va se concentrer au changement les plus important



Script controller:

Le mouvement

Premier changement à remarquer est que le fantôme va ignorer les collision du map, ceci est configuré dans unity, pas dans le script. De plus, vu qu'il lévite on enlève la gravité .

Pour introduire le développement du mouvement du fantôme, j'ai initié le processus en cherchant à implémenter une nouvelle méthode de déplacement, basée sur un patrouillage en mouvement parabolique. Au début, je me suis retrouvé face à l'incertitude quant à la manière de reproduire un tel mouvement. Après une recherche approfondie sur internet pour trouver des solutions, j'ai finalement découvert les courbes de Bézier. Cette découverte m'a donné de nouvelles connaissances en matière de conception de mouvements en courbe. Les courbes de Bézier m'ont offert une approche prometteuse pour créer des trajectoires en mouvement parabolique.

Découverte de Courbes de bézier

Les courbes de Bézier sont des outils mathématiques utilisés en infographie et en conception assistée par ordinateur pour créer des formes et des trajectoires en courbe. Elles sont définies par un ensemble de points de contrôle qui déterminent leur forme et leur

direction. Les courbes de Bézier peuvent être de différents ordres, allant de linéaires à cubiques, en fonction du nombre de points de contrôle utilisés. En utilisant des algorithmes spécifiques, ces courbes peuvent être calculées et dessinées de manière précise.

Dans le domaine de la création de jeux vidéo, les courbes de Bézier sont souvent utilisées pour animer des mouvements fluides et naturels, notamment pour le déplacement des personnages, des caméras ou des objets. Cela permet de créer des animations plus naturelles et plus esthétiques, tout en offrant un contrôle précis sur la vitesse et le rythme des mouvements.

Pour implémenter le mouvement parabolique de mon fantôme, j'ai choisi d'utiliser une courbe de Bézier quadratique. Au départ, j'ai créé de nouvelles variables pour stocker les différents paramètres nécessaires à l'équation de Bézier, à savoir p0, p1, p2 (qui représentent des vecteurs de position) et t (qui représente le temps). Dans la fonction de mouvement, au lieu de simplement modifier la vitesse du rigidbody, j'ai intégré l'équation de Bézier pour calculer la position du fantôme à chaque instant. Ainsi, le fantôme suit une trajectoire parabolique définie par la courbe de Bézier, ce qui lui confère un mouvement fluide et naturel.

Pour garantir que le fantôme reste à l'intérieur des limites de son parcours, j'ai mis en place une fonction supplémentaire appelée "setControlPoints", qui est appelée dans la fonction getDir où les limites sont vérifiées. Cette fonction permet de gérer les vecteurs de position utilisés dans l'équation de la courbe de Bézier, en veillant à ce que le fantôme reste dans les limites spécifiées. En combinant ces techniques, j'ai réussi à créer un mouvement parabolique réaliste pour le fantôme.

L'attaque

Pour coder l'attaque du fantôme, j'ai commencé par obtenir l'objet joueur comme variable dans le script d'interaction, ce qui m'a permis de sauvegarder sa position à tout moment. Contrairement à un script d'attaque conventionnel, l'attaque du fantôme se réalise en poursuivant le joueur jusqu'à le toucher, ce qui ajoute une dynamique intéressante au gameplay. Pour implémenter cette fonctionnalité, j'ai développé une nouvelle méthode pour calculer à chaque frame le vecteur direction le plus proche du joueur, en se concentrant sur les coordonnées x et y qui varient de -1 à 1.

Dans ma première tentative, j'ai utilisé un tableau de vecteurs représentant ce que je croyais être les directions possibles, telles que (-1,-1), (-1,0), (0,-1), (0,0), (1,0), (0,1), (1,1). Cependant, le résultat obtenu ne satisfaisait pas mes attentes car le mouvement n'était pas fluide. Pour améliorer cela, j'ai décidé de me tourner vers les fonctions trigonométriques Cosinus et Sinus pour tester chaque vecteur direction de manière plus précise. Au lieu d'utiliser des entiers (-1, 0, et 1), j'ai cherché à obtenir des valeurs réelles de -1 à 1, en passant par 0. Grâce à l'utilisation de Cosinus et Sinus, j'ai réussi à obtenir un mouvement plus fluide et plus naturel pour l'attaque du fantôme, ce qui a considérablement amélioré son comportement et sa capacité à poursuivre efficacement le joueur dans le jeu.

La mort et destruction de l'objet

Dans le concept du jeu, j'ai pris la décision que le personnage du fantôme soit immortel. Même s'il peut être vaincu temporairement, il ne se détruit pas définitivement. Au lieu de cela, il se ranime après quelques secondes. Cette approche a été choisie pour ajouter un élément de tension et de stratégie au gameplay, incitant les joueurs à trouver des moyens d'échapper au fantôme plutôt que de simplement le vaincre.

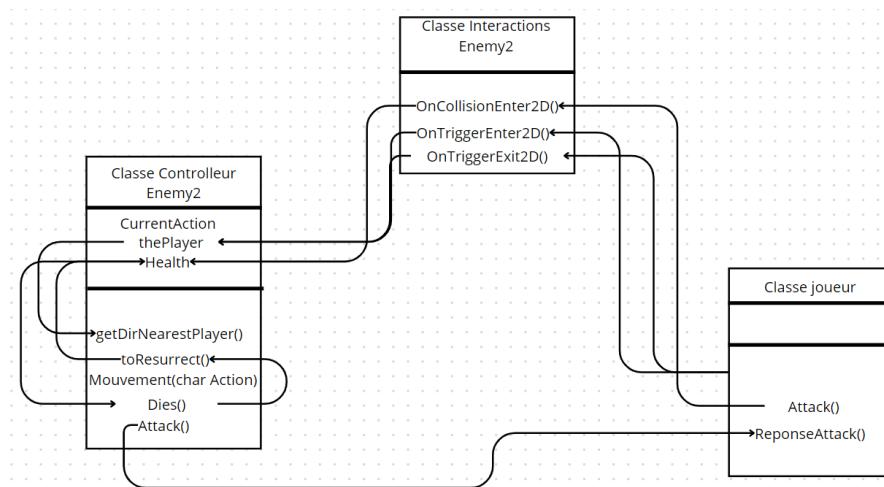
Lorsque le fantôme est vaincu, son collider devient un trigger et une animation d'effacement est déclenchée pour indiquer l'endroit où il a été vaincu. Cette animation reste en place pendant un court laps de temps pour que les joueurs ne soient pas pris au dépourvu lors de la réapparition du fantôme. Cette mécanique permet de maintenir une pression constante sur les joueurs, les obligeant à rester sur leurs gardes et à rechercher activement des moyens d'éviter le fantôme plutôt que de le confronter directement. Ainsi, l'objectif principal de cet ennemi est de créer une sensation d'urgence, renforçant ainsi l'aspect stratégique du jeu.

Script interactions:

Détection joueur

Le fantôme a un grand CircleCollider configuré en mode trigger pour détecter la présence du joueur. Lorsque le joueur entre dans cette zone, la fonction `OnTriggerEnter` est déclenchée, ce qui déclenche l'attaque du fantôme dans le script contrôleur en modifiant la variable "currentAction". Pendant que le fantôme est en phase d'attaque, son CircleCollider est agrandi pour augmenter la zone de poursuite et empêcher le joueur de s'échapper trop facilement. Ainsi, le joueur est contraint de faire face au danger et de trouver des moyens astucieux pour éviter l'attaque du fantôme.

Lorsque le joueur parvient à s'échapper de la zone de détection du fantôme, la fonction `OnTriggerExit` est déclenchée. À ce moment-là, le fantôme perd la localisation du joueur et reprend sa position de garde, ce qui permet de créer une dynamique de poursuite et d'évasion fluide et immersive. Ce système de détection contribue à maintenir un niveau élevé de tension entre le joueur et le fantôme.



Démon Volant:



Script controller:

Le mouvement

Pour gérer le mouvement du Démon volant, j'ai adopté une approche similaire à celle du squelette, en lui permettant de se déplacer horizontalement jusqu'à atteindre les limites définies par les vecteurs déjà annoncés avant. Cependant, la principale différence réside dans le fait que le Démon volant se situe et apparaît toujours dans le ciel. Dès son apparition, le Démon volant est déjà kinématique, ce qui signifie qu'il peut se déplacer sans être affecté par les lois de la physique telles que la gravité.

Contrairement au squelette, le Démon volant n'a pas besoin de gérer les cas où il pourrait rencontrer un obstacle sur son chemin, ni de craindre de tomber d'une plateforme, car il évolue dans les airs. Son mécanisme de détection du sol, habituellement utilisé pour détecter les collisions avec le sol, ne sera utilisé que pendant sa mort.

L' attaque

instantiates a prefab, explication prefab

La méthode d'attaque du Démon volant est unique et distincte de celle des autres ennemis. Contrairement à certains personnages qui utilisent des scripts d'attaque ou des boîtes d'attaque pour infliger des dégâts au joueur, le Démon volant adopte une approche

différente. Il n'a pas besoin d'un script d'attaque ni de boîtes d'attaque. Au lieu de cela, il utilise une capacité spéciale : il fait apparaître un projectile depuis sa bouche, dirigé vers le joueur afin de le toucher. Cette attaque est soumise à un temps de recharge, ce qui signifie qu'il y a un délai entre chaque utilisation.

Pendant qu'il attaque, le Démon volant continue de se déplacer de manière fluide et constante, ne s'arrêtant pas dans sa trajectoire. Toutefois, il cesse de lancer des projectiles lorsque le joueur s'éloigne trop, bien que sa vigilance ne diminue pas pour autant : une fois qu'il a repéré le joueur, il ne l'oublie pas, si le joueur se rapproche de nouveau après avoir déjà été aperçu une première fois l'ennemi relancera ces attaques. Cette mécanique ajoute une dimension stratégique au jeu, obligeant les joueurs à prendre en compte le comportement attentif du Démon volant et à ajuster leur approche en conséquence pour éviter ses attaques.

Les prefabs

Le projectile utilisé par le démon volant est un prefab. En Unity, les prefabs sont des modèles préfabriqués des GameObjects d'un jeu pré-configuré, qui peuvent être réutilisés à plusieurs reprises dans différentes parties de votre projet. Ces prefabs peuvent contenir des objets, des personnages, des scripts, des composants et d'autres éléments nécessaires au jeu.

Une fois qu'un prefab est créé dans Unity, il peut être instancié dans une scène de jeu autant de fois que nécessaire. L'instanciation d'un prefab se fait en utilisant la fonction `Instantiate()`, qui crée une copie du prefab spécifié dans la scène en cours. Lors de l'instanciation, il est possible de définir la position, la rotation et d'autres paramètres de l'objet instancié selon les besoins du jeu. Ainsi c'est comment le Démon volant va faire pour continuer à produire des projectiles

La mort et destruction de l'objet

Pour la mort et la destruction du Démon Volant, j'ai opté pour une méthode similaire à celle du squelette, mais avec quelques ajustements pour s'adapter à ses particularités aériennes. Lorsque le Démon Volant est vaincu, son processus de décès suit le même schéma que celui du squelette. Cependant, une fois qu'il est mort, son `rigidBody` devient dynamique pour prendre en compte la gravité, ce qui lui permet de tomber vers le sol. Pour détecter le moment où il atteint le sol, je remets en place la détection de collision avec le sol déjà annoncé avant. Une fois que le Démon Volant touche effectivement le sol, le processus de destruction commence, le faisant disparaître de la scène.

L'aspect le plus important dans cette gestion de la mort du Démon Volant réside dans la synchronisation des animations. Il est essentiel que les animations de la mort, de la chute et de la destruction soient parfaitement synchronisées pour offrir une expérience visuelle fluide et réaliste. Cela nécessite une coordination précise entre les différentes animations et les événements déclencheurs.

Script interactions:

Détection joueur

Pour le Démon Volant, j'ai mis en place un grand BoxCollider en mode déclencheur (trigger) qui s'étend jusqu'au sol, comme illustré dans l'image précédente. Lorsque le joueur rentre dans cette zone, j'utilise la fonction `OnTriggerEnter` pour détecter cette interaction. Cependant, contrairement à d'autres ennemis qui pourraient perdre la trace du joueur lorsqu'il sort de leur zone de détection, le Démon Volant conserve en permanence la trace du joueur dans son script.

Pour cette raison, je n'utilise pas la fonction `OnTriggerExit` pour gérer la sortie du joueur de la zone de détection du Démon Volant. Au lieu de cela, lorsque le joueur est enregistré, je désactive simplement le BoxCollider en mode déclencheur du Démon Volant. Cela permet de garantir que le Démon Volant reste activement engagé dans la traque du joueur tant qu'il est dans les alentours, tout en évitant les problèmes potentiels de perte de trajectoire ou d'oubli du joueur par l'ennemi. En utilisant cette approche, je crée un système de détection du joueur continu pour le Démon Volant.

Projectile

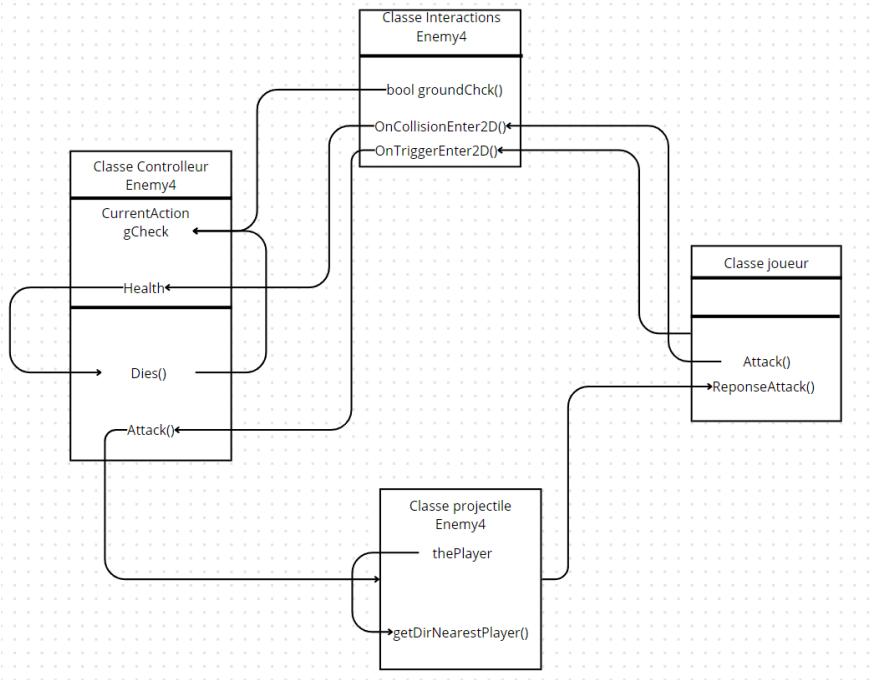
Script controller:

Le mouvement

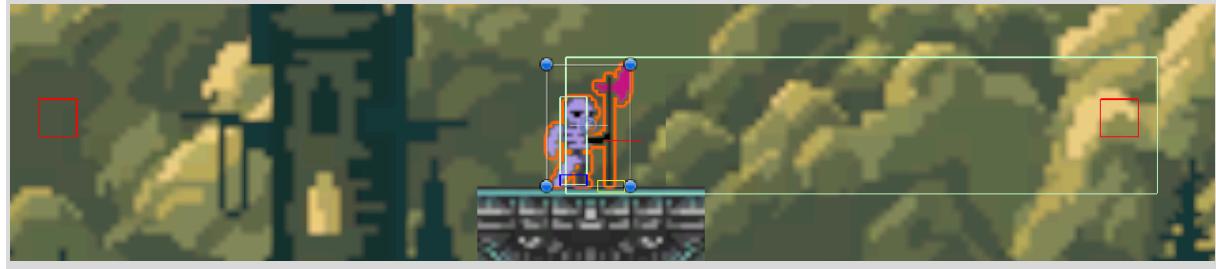
Dès son apparition, le projectile est informé de la position actuelle du joueur, ce qui lui permet de démarrer son mouvement en direction de sa cible. Pour ce faire, le projectile utilise la même méthode employée par le fantôme pour déterminer le vecteur direction le plus proche du joueur, assurant ainsi une poursuite efficace.

Cependant, afin de maintenir un équilibre dans le jeu et de fournir au joueur une opportunité de réagir, une limite de distance est définie pour le projectile. Lorsque le projectile se rapproche du joueur en dessous d'une certaine distance, il cesse de le poursuivre et continue sa trajectoire, permettant au joueur de tenter d'esquiver le projectile. Cette mécanique ajoute une dimension stratégique au jeu, incitant les joueurs à anticiper les mouvements du projectile et à esquiver pour éviter les attaques.

En outre, pour éviter toute accumulation excessive d'objets sur la scène, chaque projectile est configuré pour se détruire automatiquement après un laps de temps prédéfini, généralement fixé à cinq secondes.



Squelette violet:



Script controller:

Le mouvement

Pour le mouvement de la variation du squelette, le squelette violet, j'ai adopté une approche basée sur l'héritage des fonctionnalités du squelette de base. En tant que classe enfant du squelette normal, le squelette violet hérite la majorité de ses fonctionnalités de base, y compris son schéma de mouvement. Par conséquent, le mouvement du squelette violet reste essentiellement le même que celui du squelette de base.

Cependant, ce qui différencie le squelette violet réside principalement dans sa façon d'attaquer et sa vitesse de mouvement et d'attaque. Contrairement au squelette de base, le squelette violet est plus rapide à la fois dans ses animations et son patrouillage. Cette augmentation de vitesse lui permet d'être plus agressif dans ses interactions avec le joueur, augmentant ainsi le niveau de défi et d'intensité du jeu.

L' attaque

j'ai introduit une série de mécanismes uniques visant à le distinguer de son homologue, le squelette normal. La principale différence réside dans le champ de vision élargi du

squelette violet, lui permettant de détecter le joueur à une plus grande distance. Lorsqu'il perçoit le joueur, le squelette violet enregistre instantanément sa localisation, se préparant ainsi à l'attaquer.

Une fois le joueur repéré, le squelette violet sprinte vers le point où il l'a aperçu, se précipitant avec une agilité accrue pour engager l'attaque. Une fois arrivé à destination, il lance son coup. Après avoir effectué son attaque, le squelette violet reprend sa patrouille, oubliant la localisation précédente du joueur. Ceci ajoute une difficulté plus élevée rendant l'ennemi plus imprévisible et redoutable pour le joueur. En résumé, l'attaque du squelette violet combine une détection du joueur avec des mouvements rapides et attaques rapides, créant une expérience de jeu rapide.

Dégâts et temp d'invulnérabilité

Étant donné que le squelette violet possède plus d'un point de vie et ne meurt pas immédiatement après avoir subi des dégâts, il est essentiel de mettre en place une méthode pour gérer sa réponse initiale à ces dommages. Pour ce faire, j'ai créé un mécanisme qui accorde temporairement au squelette violet une période d'invulnérabilité après avoir subi un premier dégât.

Cette période d'invulnérabilité est conçue pour protéger le squelette violet d'une série de dégâts rapides qui pourraient le mettre en danger de manière injuste ou déséquilibrée. Pendant cette fenêtre de temps, le squelette violet est immunisé contre les attaques supplémentaires, lui permettant de se remettre de l'impact initial et de reprendre un comportement normal.

La mort et destruction de l'objet

La mort et de la destruction du squelette violet reste fidèle au modèle établi par son parent, le squelette normal. Lorsque le squelette violet est vaincu, le processus de décès qui s'ensuit est le même que celui du squelette de base. Une fois que ces points de vie atteignent zéro, le squelette violet entre dans son état de mort, et après quelques secondes il se détruit de la scène.

Script interactions:

Détection joueur

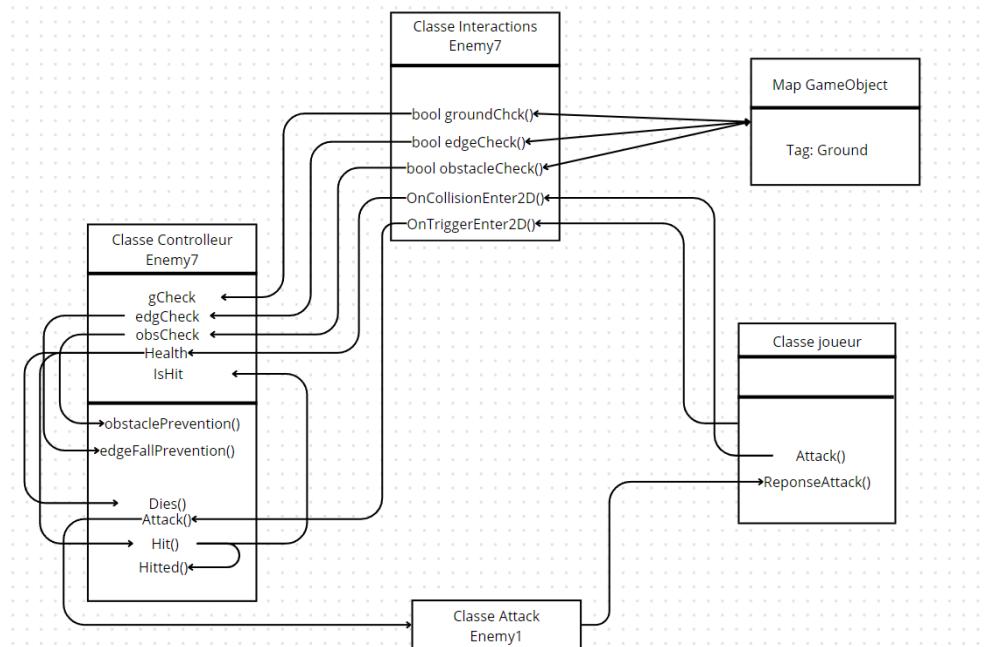
Le squelette violet hérite du même système de détection que son parent, le squelette normal, mais avec quelques ajustements pour répondre à ses caractéristiques uniques. Le squelette violet dispose d'un trigger plus étendu, lui permettant de repérer le joueur à une distance plus grande. Lorsque le joueur entre dans la zone de détection du squelette violet, un mécanisme est enclenché pour enregistrer la position du joueur, à condition qu'aucun obstacle ne soit présent entre eux.

Ce qui différencie le squelette violet est son processus de vérification supplémentaire avant d'enregistrer la position du joueur. Une fois que le joueur est détecté, le squelette violet effectue un raycast entre sa propre position et celle du joueur pour détecter la présence d'obstacles sur le chemin. Si aucun obstacle n'est détecté, le squelette violet enregistre alors la position du joueur et engage son attaque. En revanche, si un obstacle est détecté, le

squelette violet conserve sa position actuelle et continue sa patrouille, évitant ainsi de poursuivre le joueur à travers des obstacles infranchissables.

Script attaque:

Le Script d'attaque est le même que pour le squelette normal.



Démon Volant rouge:



Script controller:

Le mouvement

Pour développer le mouvement de la variation Démon volant rouge, En tant que classe enfant du Démon volant normal, le Démon volant rouge conserve la plupart des fonctionnalités de mouvement de son parent. Ainsi, son schéma de mouvement global reste essentiellement le même que celui du Démon volant normal.

Les principales différences se situent dans la façon d'attaquer et la vitesse du mouvement du Démon volant rouge. Contrairement au Démon volant normal, le Démon volant rouge est plus rapide à la fois dans ses attaques et ses déplacements. Cette accélération dans son comportement lui permet d'engager le joueur de manière plus agressive, augmentant le niveau de défi du jeu.

L' attaque

Pour le Démon volant rouge, j'ai choisi une approche distinctive par rapport à son parent. Contrairement à ce dernier, le Démon volant rouge a des attaques à courte portée, ce qui signifie qu'il engage le joueur avec une agressivité accrue lorsque celui-ci est aperçu. L'attaque du Démon volant rouge se manifeste par un mouvement caractéristique : il se balance depuis les cieux vers le joueur avec un mouvement parabolique, créant ainsi une trajectoire d'attaque redoutable.

Pour réaliser cette attaque parabolique, j'ai fait appel à la courbe de Bézier quadratique, qui permet au Démon volant rouge de calculer une trajectoire précise en fonction de la position du joueur. Une fois qu'il a localisé sa cible, le Démon volant rouge s'élance vers elle. Il est important de noter que l'attaque du Démon volant rouge est soumise à un temps de refroidissement, ce qui lui donne un attaque équilibrée.

Tout comme son parent, une fois qu'il a repéré le joueur, le Démon volant rouge ne le laisse pas échapper jusqu'à ce que l'un d'eux soit vaincu. Pendant le temps de refroidissement de son attaque, le Démon volant rouge continue de poursuivre le joueur depuis les airs, cherchant à maintenir sa position la plus proche possible en vue de sa prochaine offensive.

Dégâts et temp d'invulnérabilité

Même que pour le squelette violet

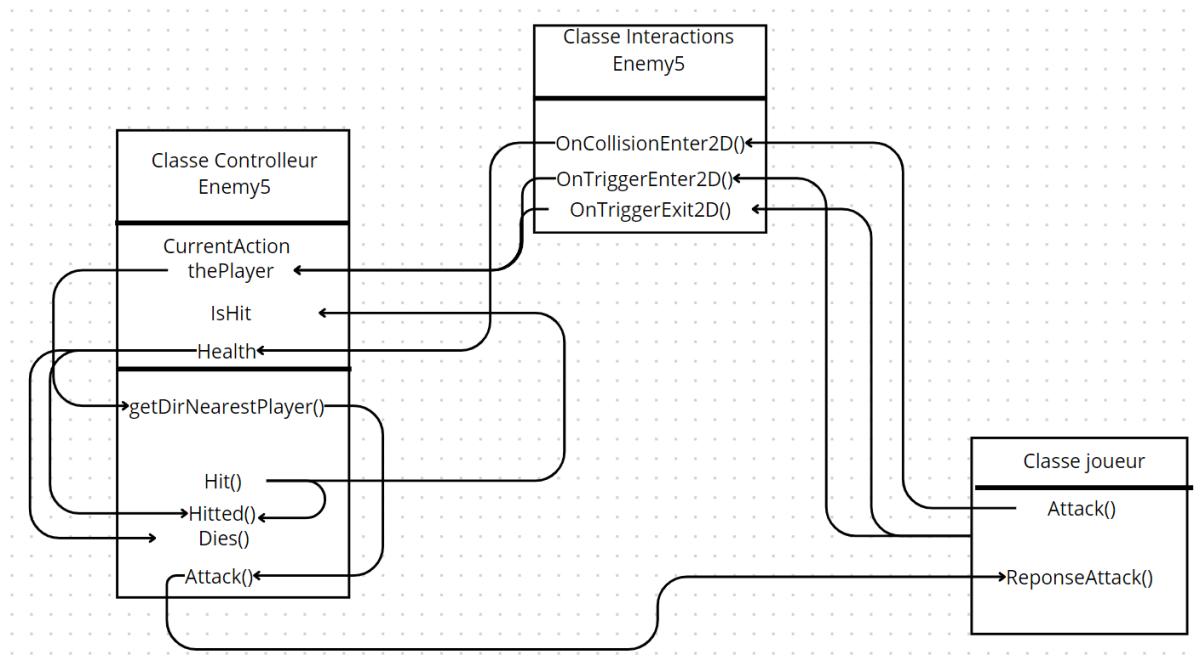
La mort et destruction de l'objet

Même que son père

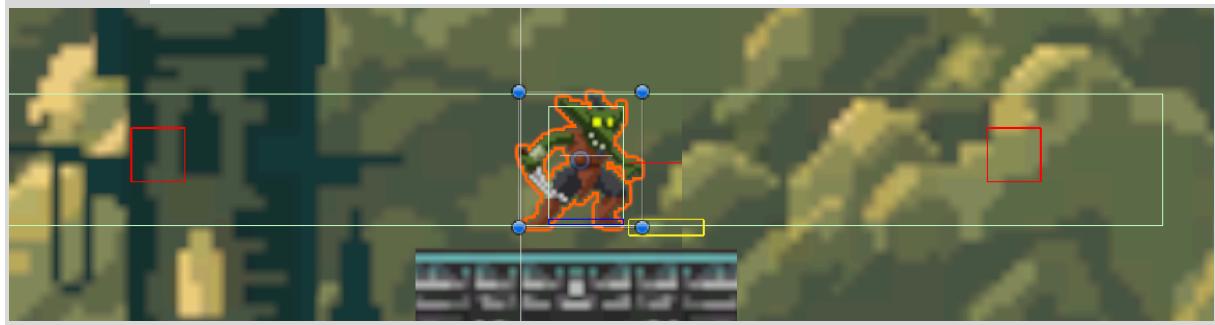
Script interactions:

Détection joueur

La principale différence par rapport à son parent réside dans la taille de son Trigger BoxCollider, qui est ajustée pour correspondre aux caractéristiques du Démon volant rouge. Une fois que le joueur pénètre dans cette zone de détection étendue, il est enregistré de manière permanente par le Démon volant rouge. Il maintient une vigilance constante une fois qu'il a repéré sa cible. Cette stratégie garantit que le Démon volant rouge reste engagé dans sa poursuite du joueur, lui permettant de maintenir une pression constante.



Gobelins:



Script controller:

Le mouvement

Pour développer le mouvement du Gobelin, j'ai choisi de baser son comportement sur celui du squelette normal, en conservant les principes de mouvement de base déjà établis. Ainsi, le Gobelin se déplace de la même manière que le squelette normal, avec des déplacements horizontaux et des actions de patrouille dans des zones prédéfinies. Cependant, ce qui différencie le Gobelin est son attaque, qui est conçue pour être plus complexe, dynamique et intelligente que celle des autres ennemis.

Le attaque

L'attaque du Gobelin est le point central de sa conception, avec une approche plus sophistiquée et adaptative pour cibler le joueur de manière stratégique. Lorsque le Gobelin détecte la présence du joueur, il entre en mode d'attaque, cherchant à s'approcher

suffisamment pour infliger des dégâts. Ce qui distingue vraiment le Gobelins des autres ennemis, c'est sa capacité à suivre le joueur même lorsque ce dernier saute sur une plateforme ou change de plateforme.

Pour réaliser cela, j'ai développé un système dans un nouveau script appelé "Jump" qui communique avec le script contrôleur du Gobelins. Le Gobelin est programmé pour suivre les déplacements du joueur de manière dynamique, en ajustant sa trajectoire pour rester en phase avec les mouvements du joueur. Cette capacité lui permet de poursuivre le joueur avec une grande précision, même lorsqu'il effectue des actions acrobatiques telles que le saut entre les plateformes.

Dégâts et temp d'invulnérabilité

Pour assurer la résilience du Gobelins face aux attaques, j'ai mis en place un système similaire à celui utilisé pour le squelette violet et le Démon volant rouge. Comme ces ennemis, le Gobelin est doté de plusieurs points de vie, ce qui lui permet de survivre à plusieurs dégâts avant d'être vaincu. Cependant, afin de gérer sa réponse à un premier dégât, j'ai implémenté un mécanisme d'invulnérabilité temporaire.

Après avoir été touché, le Gobelin entre dans un état d'invulnérabilité pendant un certain laps de temps. Pendant cette période, il devient temporairement insensible aux attaques. Ce qui distingue le Gobelin des autres ennemis est sa réaction à cet état d'invulnérabilité. Contrairement à une simple immunité aux dégâts, le Gobelin est programmé pour être repoussé en arrière selon le chemin d'une courbe de Bézier quadratique.

Plutôt que de simplement résister aux attaques pendant un court laps de temps, le Gobelin réagit de manière à avoir une réponse physique à l'impact subi.

La mort et destruction de l'objet

De la même façon que les démons volants.

Script interactions:

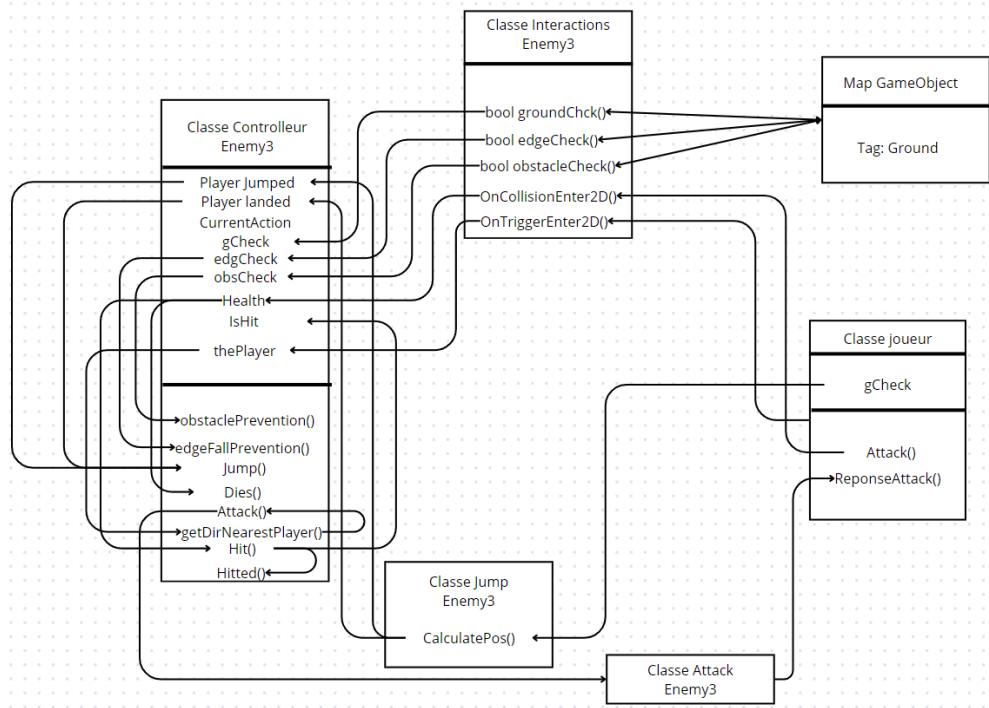
Comme pour le squelette violet, il vérifie aussi s'il y a des obstacles avant de démarrer sa poursuite.

Script Jump:

On arrive à la partie la plus importante de cet ennemi. Le script de saut du Gobelins offre une fonctionnalité particulière qui lui permet de suivre le joueur de manière dynamique à travers les différentes plateformes du jeu. Lorsque le Gobelin est en mode de poursuite du joueur, le script jump prend en compte la position du joueur grâce à une variable qui enregistre le gameObject correspondant au personnage jouable. Cela permet au Gobelin d'accéder à la position actuelle du joueur ainsi qu'au script de son personnage.

Pour calculer le saut du Gobelin, le script jump surveille les mouvements du joueur et enregistre la position de son premier saut, détectée grâce à une variable appelée ground check. Lorsque le joueur retombe au sol après ce premier saut, le script vérifie si sa position d'atterrissement est à une hauteur supérieure à celle du premier saut. Si tel est le cas, le

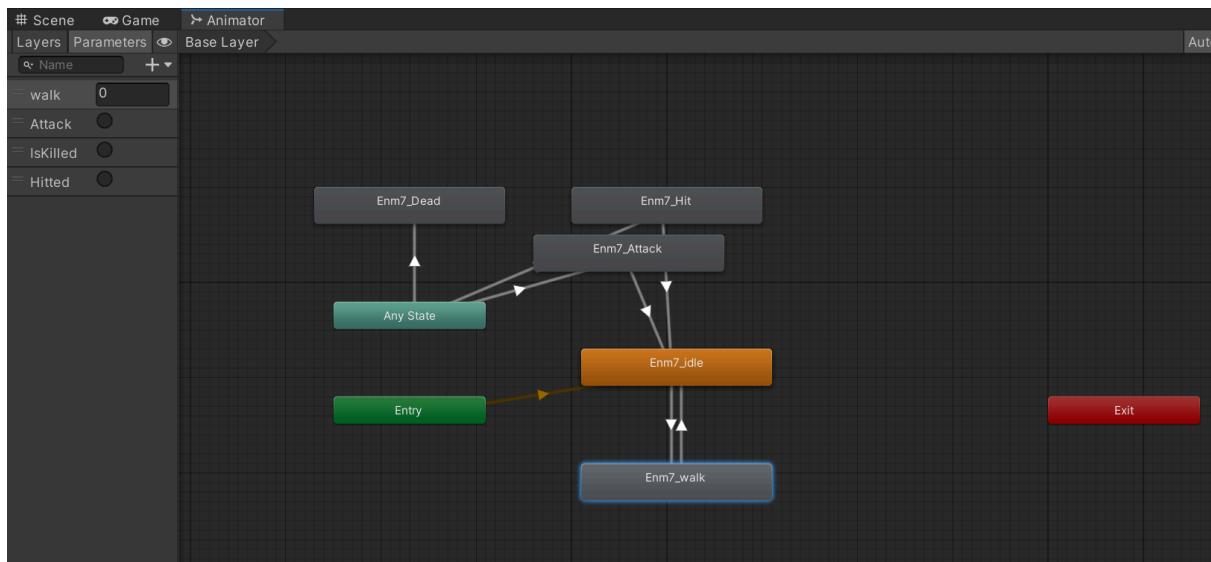
Gobelin enregistre également cette position d'atterrissement, sinon il oublie les deux positions et continue sa poursuite.



Une fois que le Gobelin a réussi à enregistrer ces deux positions, il utilise la formule de la courbe de Bézier quadratique pour créer une trajectoire de saut entre les deux points. Cette trajectoire est conçue pour permettre au Gobelin de suivre le joueur même s'il se déplace vers une autre plateforme ou une autre zone élevée.

Gestion animation

Fenêtre pour gérer les animations:



(exemple avec l'ennemi 7)

(le composante animator va utiliser cette fenêtre)

Utiliser la composante animator et la fenêtre pour créer des conditions à remplir, gérer et synchroniser les animations avec les actions dans le jeu.

Développement de la map :

Pour pouvoir avoir une map dans laquelle le protagoniste pourra évoluer (se battre contre des ennemis et utiliser des différentes mécaniques), j'ai dans un premier temps dû créer un "tilemap".

Un tilemap est une technique couramment utilisée dans les jeux vidéo, notamment dans les jeux 2D, pour créer des niveaux ou des environnements en utilisant une grille de tuiles (tiles) pré-dessinées. Unity, un moteur de jeu populaire, offre un support intégré pour les tilemaps, ce qui simplifie grandement le processus de création d'environnements 2D.

Voici comment l'utiliser :

1) La création :

Dans Unity, il faut utiliser la modélisation 2D.

Allez dans 'GameObject' -> '2D Object' -> 'Tilemap' .

2) Sélection d'une tuile :

Une fois que l'on a créé la tilemap, on doit sélectionner les tuiles qui seront utilisées lors du projet.

On peut importer en tant que sprite (ça sera notre cas ici) ou même créer ses propres tuiles.

Dans mon cas, j'ai dû grâce à l'outil palette découper les sprites pour les isoler un par un, pour pouvoir les utiliser indépendamment des autres.

3) Peindre sur le tilemap :

Avec l'outil 'brush', cliquez et faites glisser pour peindre les tuiles sur le tilemap. On peut remplir de grande zone ou dessiner des formes plus complexes en fonction de nos besoins.

4) Collisions et propriétés :

On peut également définir des collisions sur nos tuiles pour déterminer comment les objets interagissent avec elle. Cela peut être en utilisant le 'tilemap collider' ou en assignant des propriétés spécifiques aux tuiles dans le tilemap.

Certaines tuiles peuvent avoir des propriétés spéciales , comme des plateformes mobiles, des portes, etc. On peut définir ces propriétés dans l'éditeur Unity.

5) Scripting :

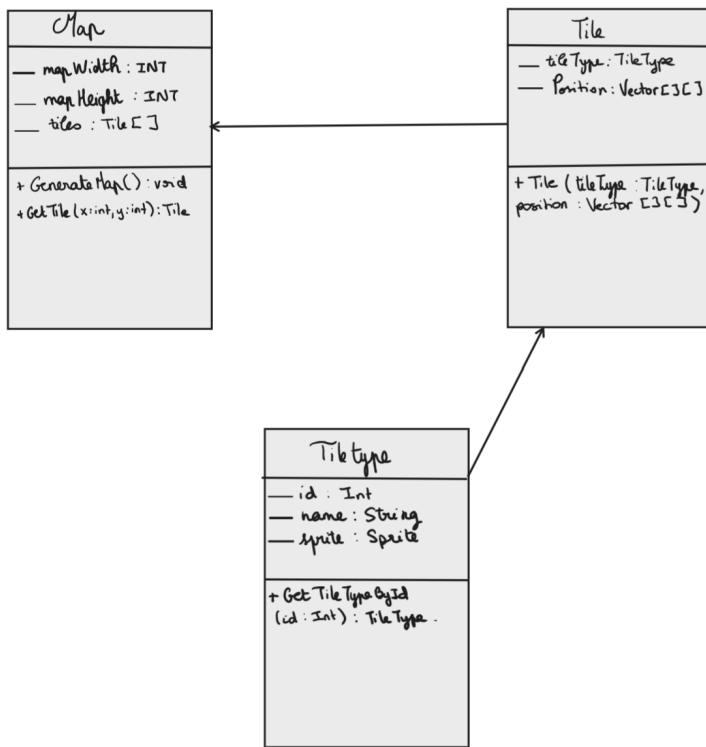
On peut également interagir avec notre tilemap via des scripts Unity. On peut modifier dynamiquement les tuiles en fonction des actions du joueur.

impression écran des caractéristiques de la tilemap :

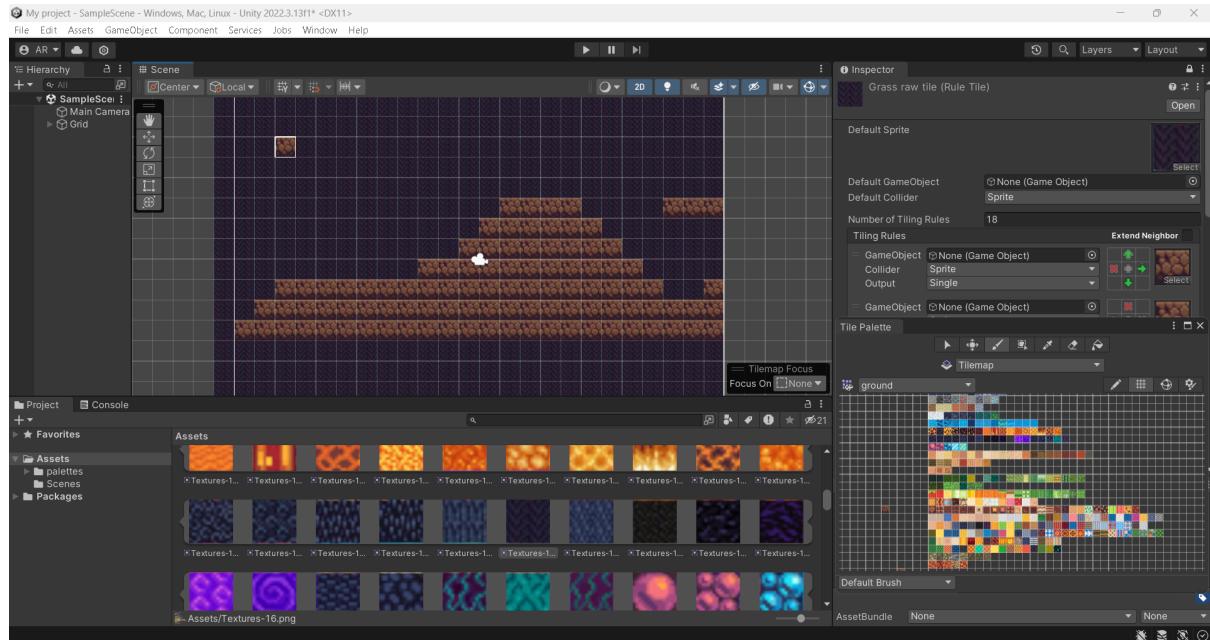


Il s'agit là de la partie théorique, passons à la pratique !

Conceptualisation avec un diagramme de la création de la map :



Résultat du développement de la map :



Voici les résultats obtenus pour la conceptualisation de la map du jeu sur Unity.

La map devait posséder plus de caractéristiques, cependant par manque de temps je n'ai malheureusement pas pu implémenter toutes les fonctionnalités souhaitées.

Les idées non implémentées sont : Des plateformes mouvantes (de gauche à droite, ou de haut en bas, et des plateformes qui réagissent au personnage principal pour tomber une fois qu'il passe dessus). Ces fonctions auraient permis de créer différentes énigmes et une autre perspective de gameplay pour le joueur.

Conclusion:

Larmat Jean:

Dans le cadre de ce projet, j'ai eu l'opportunité de plonger dans l'univers de Unity pour la première fois de ma vie. Une grande partie de mon temps a été consacrée à apprendre les tenants et aboutissants de cette plateforme, des différentes composantes à la gestion des fenêtres, en passant par le paramétrage et les capacités de Unity. J'ai également exploré le scripting, en comprenant la matrice des fonctions de MonoBehaviour, ce qui m'a permis de construire le jeu de manière autonome. Au fil du projet, j'ai pris conscience de la complexité cachée derrière des jeux en apparence simples, tels que les platformers. Chaque détail, demande une attention particulière et une réflexion approfondie. Cette expérience m'a donné un aperçu de la complexité et de l'ampleur du travail nécessaire à la création de nouveaux jeux par exemple triple A impliquant souvent des centaines de personnes travaillant en collaboration.

Malgré les défis rencontrés, je suis satisfait du travail accompli, bien que j'aurais aimé pouvoir consacrer davantage de temps au projet. Je suis quand même convaincu que les améliorations que j'ai apportées à mes compétences en codage valent le temps investi. Tout le code a été conçu par moi, en m'appuyant sur les ressources disponibles en ligne pour comprendre les concepts et les techniques nécessaires. Cette approche m'a permis de développer mon propre code, avec une implication minimale de copier-coller.

Ce projet m'a également enrichi sur le plan professionnel, en m'offrant de nouveaux savoir-faire dans le domaine de l'informatique graphique, un domaine qui suscite particulièrement mon intérêt. Cependant, je reconnaiss que des améliorations sont possibles, notamment en ce qui concerne la communication au sein du groupe et l'organisation du code. Je me suis parfois retrouvé à écrire des lignes de code sans une structuration adéquate, ce qui a rendu la navigation dans mon propre code moins fluide. Dans l'avenir, je m'efforcerai de mieux gérer la programmation orientée objet, en envisageant davantage l'héritage et le polymorphisme pour une meilleure architecture logicielle.

Ricquet Alicia

L'idée principale est plus de temps pour pouvoir faire un meilleur rendu. Pour ma part le projet a été un vrai défi à relever car je n'avais pas de connaissances dans le domaines du jeux-vidéo, de Unity ou du langage C#. Je suis ravie d'avoir pu apprendre à maîtriser de nouvelles technologies, mais je suis déçue de ne pas avoir pu faire plus pour le projet car la prise en main du tout a monopolisé une majeure partie de mon temps.

J'ai également pu constater que travailler sur plusieurs choses en même temps (ici les cours et le projet en parallèle) était complexe mais m'a poussée à me surpasser.

D'un point de vue technique, une amélioration et une implémentation des éléments restants pour rajouter du challenge pour le joueur serait la prochaine étape à suivre si l'on souhaite travailler davantage sur le projet et si le temps nous l'accorde.

Pour conclure le projet m'a apporté des connaissances techniques pour mon avenir professionnel, mais cela m'a également permis de développer des qualités comme l'organisation et le travail en équipe qui me seront aussi bien utiles dans ma vie personnelle que professionnelle.

Marechal Baptiste

Travailler sur ce projet a été une expérience extrêmement enrichissante pour moi, cela m'a permis d'acquérir un grand nombre de compétences et de connaissances, tant sur le plan technique que sur le plan personnel. J'ai appris à surmonter des obstacles et à travailler en équipe de manière plus efficace car je n'ai pas pu avoir beaucoup de possibilités de travailler en équipe sur un projet de cette taille avant ce projet.

Références:

<https://ansimuz.itch.io/gothicvania-church-pack>:

<https://ansimuz.itch.io/gothicvania-cemetery>

Vidéo “TILEMAPS in Unity” :

https://www.youtube.com/watch?v=ryISV_nH8qw&t=1s

Vidéo “Create Dynamic Tilemaps in SECONDS - Unity Tutorial” :

https://www.youtube.com/watch?v=g83_gwEO0kM&t=2s

Vidéo “Unity Tutorial - How to Make Moving Platform in Unity | Unity Tutorial for Beginners” :

https://www.youtube.com/watch?v=vua2a_Z3zIY

Vidéo “How to Make a Camera System (Like Hollow Knight's) in Unity using Cinemachine | 2D Tutorial” :

https://youtu.be/9dzBrLUIF8g?si=luG-thO_E0XXvG5U