UNIVERSITÉ CLAUDE BERNARD LYON 1
UCBL LYON 1

DÉPARTEMENT INFORMATIQUE: LICENCE 3EME ANNÉE
2020/2021

Projet de substitution au stage / Non effectué dans l'entreprise ECAB

Création de jeu style RPG sous Unity 3D

SOMMAIRE

I – Introduction

Pour mon stage de troisième année de Licence Informatique, en raison de la situation sanitaire, je n'ai pas pu trouver de proposition de stage dans les temps. J'ai donc effectué un projet de remplacement sur mon propre ordinateur. Pour ce rapport, voici une entreprise type qui aurait pu m'accueillir pour réaliser mon projet: l'entreprise imaginaire ECAB à Villeurbanne.

J'ai donc effectué un stage dans l'entreprise ECAB située à Villeurbanne. Cette entreprise se concentre sur le développement de jeux vidéos modélisés en 3D à l'aide d'Unity 3D. Les jeux sont développés en C#.

La mission qui m'a été attribuée était de développer une variante d'un jeu de style RPG sous Unity 3D. Un jeu de style RPG correspond à un jeu dans lequel le joueur apparaît pour la première fois dans un monde au niveau 0 et avec un équipement basique. Le but du jeu sera d'éliminer des monstres où de réaliser des quêtes afin de gagner de l'expérience et un meilleur équipement.

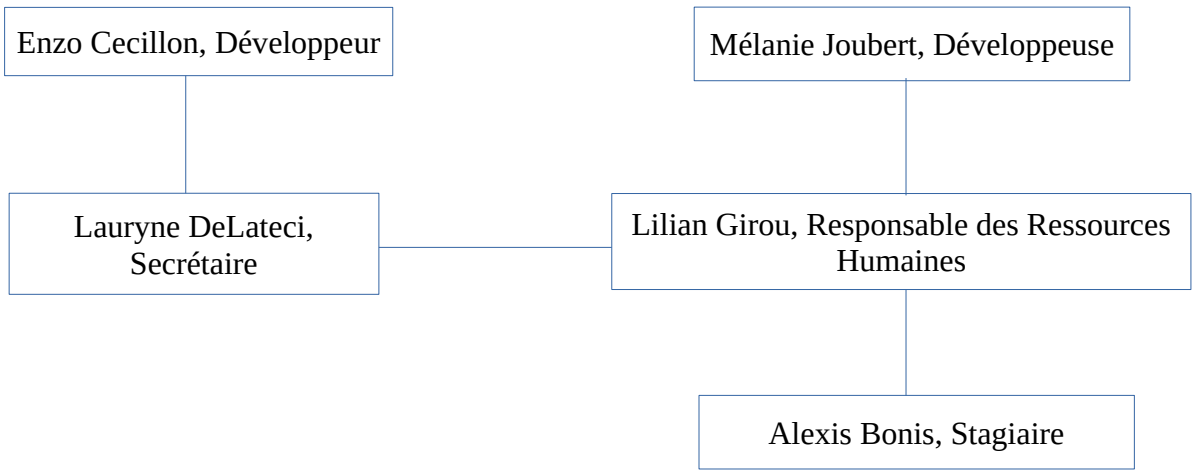
Dans un premier temps, nous verrons l'environnement professionnel. C'est à dire certaines données de l'entreprise, le service ainsi que quelques chiffres et un organigramme de l'entreprise.
Dans un second temps, nous étudierons les différents outils utilisés pour réaliser ce projet, ainsi que les méthodes de travail associées.
Ensuite, nous verrons comment la mission qui m'a été confiée s'est déroulée. Plus particulièrement les objectifs du sujet, ma méthodologie ainsi que les difficultés rencontrées.
Puis nous verrons mon retour sur l'expérience. Entre autres les compétences acquises et savoir faire, ainsi que les méthodes de travail apprises.
Avant de terminer ce rapport, nous effectuerons une conclusion comprenant un récapitulatif du sujet, le lien avec ma formation actuelle ainsi que mes perspectives d'avenir.
Et enfin, nous terminerons avec les annexes comportant les différents scripts utilisés.

II – Environnement Professionnel

Voici une entreprise qui aurait pu m'accueillir pour réaliser ce projet : L'entreprise fictive ECAB de type SAS (Société par Action Simplifiée).
Cette entreprise, située à Villeurbanne, travaille dans le domaine du développement de jeux vidéos en 3D. Le langage et le moteur graphique principalement utilisé sont le C# et le moteur Unity 3D.
Cette entreprise a notamment contribué au développement du jeu *Real Life RPG*, un jeu de type RPG se situant dans notre monde, à notre époque.
ECAB possède actuellement seulement 4 employés, dont deux développeurs, une secrétaire et un responsable des ressources humaines. L'ancienneté de l'entreprise est également seulement de 2 ans. Elle est donc encore très jeune sur l'énorme marché du jeu vidéo. Elle possède un chiffre d'affaire de 150 000€ durant l'année 2020, et possède un bénéfice de 71 % soit de 106 500€.
L'entreprise ECAB est actuellement en pleine expansion, suite au succès international de leur jeu, et commence à recruter bon nombre de personnes, ainsi que de stagiaires. En effet, ils essayent de former un maximum de personnes pour pouvoir recruter un maximum de développeur afin de répondre à la demande croissante de projets venant de plus grosses entreprises de jeu vidéo, telle que EA Games.

Je n'ai donc pas été placé dans une équipe, en raison du peu de personne composant l'entreprise actuellement. Lors de mon entretien pour le stage, le responsable m'avait précisé que je serais totalement seul, sans personne pouvant m'aider, afin de voir mon réel potentiel d'apprentissage et de programmation C#.

L'organigramme de l'entreprise est le suivant :



III – Outils utilisés / Méthode de travail

La mission qui m'a été confiée consiste a développer un jeu de style RPG sous le moteur graphique Unity 3D. J'ai donc installé le logiciel Unity sur ma machine personnelle.
Le moteur graphique Unity est gratuit et en libre accès. C'est un moteur puissant permettant de réaliser bon nombre de projets complètements différents, bien que majoritairement des jeux vidéos. En effet, j'ai déjà eu la chance d'utiliser Unity afin de faire un projet visant à créer un jeu vidéo de plateformes en 2D, ce qui m'a également facilité l'adaptation pour ce projet.

Unity supporte le JavaScript et le C#, ici nous utiliserons ce dernier. Grâce à cela, nous pouvons créer des scripts qui permettent d'animer le jeu. Par exemple ajouter des événements que le joueur peut activer, ou comme la mort du personnage lorsque sa vie arrive à 0, ou créer une intelligence artificielle pour des ennemis hostiles au joueur, même jusqu'à créer la logique du jeu elle même. Unity possède sa propre bibliothèque graphique que l'on peut importer dans les scripts en C#, ce qui permet de faire le lien entre les scripts et les différents objets du jeu. Que ce soit des objets de l'interface où des objets en 3D. Pour la création d'un jeu de style RPG, ces scripts sont donc indispensables pour gérer tous les événements possibles.

Ce moteur dispose d'énormément d'outils intégrés. Effectivement, lorsque l'on installe Unity pour la première fois, on installe l'interface de gestion de versions d'Unity, qui nous permettra de choisir quelle version nous voulons utiliser ainsi que les modules que nous voulons installer. Une fois la version voulue et les modules installés, vous pouvez lancer Unity.
Une fois lancé, il propose une interface possédant beaucoup d'outils nous facilitant la programmation des scripts, la création de scène en 3D, la personnalisation de l'UI (autrement dit ce qui correspond à l'interface), ou encore la gestion de fichiers et de hiérarchie des objets.
Pour ce qui est de la création de la scène en 3D, Unity possède deux fenêtres d'inspections, Game et Scene, qui nous permettent de visualiser directement le rendu de nos ajouts comme si nous lancions le jeu (Game), ou alors de manière globale (Scene). On peut importer des formes basiques, comme des sphères ou des cubes, directement depuis l'interface de base d'Unity. Nous pouvons également ajouter des caméras, des lumières, gérer les surfaces et reliefs du terrain, et bien sûr modifier les paramètres de chaque objets comme la taille ou la position et leur ajouter des scripts et composants.
L'interface dispose également d'un créateur d'animations à partir de modèle 3D importé dans le projet, mais aussi d'une boutique permettant de télécharger des assets, qui correspondent à des modèles 3D, 2D, de l'UI, ou même juste des scripts permettant de réaliser diverses choses. Malheureusement, la grande majorité des assets de la boutique (appelée Asset Store) est payante. Mais Unity permet l'importation d'objets 3D extérieurs au site officiel Unity et à l'Asset Store. Ce qui élargit encore plus les possibilités de création de projets avec le moteur Unity.
Ce dernier nous permet même de créer des fenêtres de paramètres modifiables directement dans le jeu. Comme par exemple modifier le volume du jeu, le volume de la musique, ou même changer la résolution pour mieux adapter le jeu en fonction de la machine sur laquelle il est lancé.

Pour mon stage j'ai été entièrement seul, je n'ai pas reçu d'aide ou de soutien. J'ai donc beaucoup cherché de l'aide sur internet lorsque j'étais bloqué. Un grand nombre de tutoriels gratuits pour novices existent, ce qui m'a grandement aidé lorsque je rencontrais des difficultés.

IV – Mission / Sujet du stage

A) Création du projet et du terrain

Tout d'abord j'ai commencé par créer le projet dans Unity, ainsi que le terrain. L'interface de gestion de versions d'Unity, Unity Hub, permet aussi de gérer les différents projets liés à notre compte Unity. Nous pouvons en créer où en importer depuis notre ordinateur. C'est donc d'ici que j'ai créé le projet ainsi que tous les fichiers de base associés.

Une fois le projet créé, j'ai lancé Unity et directement créé un dossier « Asset » pour y ranger les différents assets que je téléchargerai depuis l'Asset Store ou depuis internet. Mais lorsque l'on créé un nouveau projet, le rendu de notre jeu dans les inspecteurs Game et Scene, sont vides. En effet c'est un tout nouveau projet il n'y a encore donc rien qui n'a été créé, mis à part la caméra ou la lumière présentes de base. Pour créer le premier terrain, il faut donc créer un GameObject de type Terrain. Les GameObjects sont les différents objets 2D ou 3D que l'on peut mettre dans notre jeu, il en existe de toute sorte. Lorsque le terrain est créé, il est plat et sans texture avec une taille par défaut. C'est donc à moi de lui donner de la texture, mais aussi des reliefs et la taille que l'on souhaite. Pour la taille du terrain je ne savais pas encore quel genre de terrain j'allais faire, j'ai donc un peu agrandi la taille pour me permettre de tester plus de choses pour plus tard. J'ai ensuite utilisé l'Asset Store pour télécharger un pack de textures que j'ai directement importé dans mon projet. Ce pack de textures pourra également nous servir pour ajouter de la texture à des objets comme des arbres ou rochers par exemple. L'avantage d'Unity est que chaque texture peut être utilisé sur n'importe quel GameObject, autrement dit sur absolument tout !

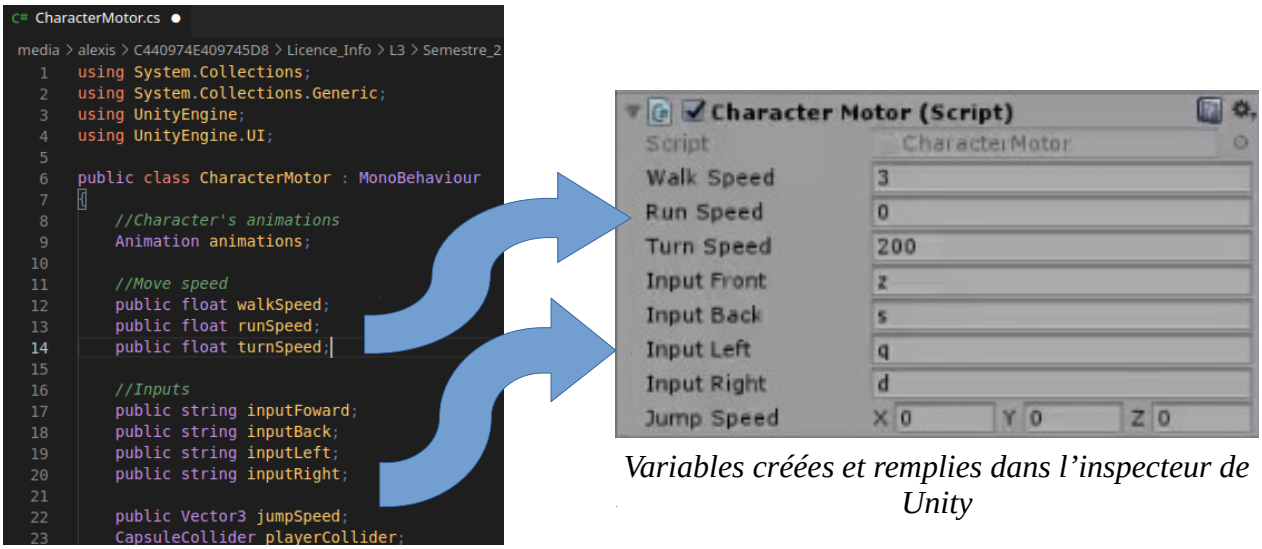
Dans les paramètres du terrain, Unity propose un outil qui permet directement de peindre le terrain, avec des textures sélectionnées au préalable. J'ai donc choisi une texture d'herbe provenant du pack téléchargé et recouvert mon terrain entièrement de cette texture pour commencer. Puis je me suis occupé des reliefs. En effet au même endroit que pour les textures du terrain, Unity dispose d'un outil permettant de modifier les reliefs en choisissant la force et la taille de notre « pinceau », de la même manière que pour les textures. Le clic gauche servant à ajouter du relief, et le clic droit servant quant à lui à en enlever. Pour m'entraîner j'ai donc décidé d'ajouter des sortes de montagnes faisant tout le tour du terrain pour empêcher le futur joueur de sortir du terrain et de tomber dans le vide. Ensuite j'ai créé une montagne au milieu du terrain que le joueur pourrait monter. La création de cette montagne fût très compliquée mais m'a permis de vraiment bien prendre en main les outils de modification de terrains. J'ai téléchargé des assets de roches et de maisons médiévales, puis j'ai mis des roches sur le terrain afin de voir le fonctionnement, et aussi une maison sur la crête de la montagne. Ce qui m'a permis de m'améliorer encore plus dans la gestion des reliefs et des textures, en ajoutant une texture de chemin en pierre allant du pied de la montagne jusqu'à la maison.

B) Création du joueur et des fonctionnalités de base

Une fois le terrain de base créé, je me suis attaqué à la création du joueur. Pour commencer j'ai téléchargé un asset d'un personnage représentant un comabatant spartiate depuis l'Asset Store. Ce personnage possède des animations de base et donc un squelette ce qui me facilitera beaucoup de choses. De plus son design correspond à un personnage principal de RPG, autrement dit un combattant médieval. Malheureusement ce personnage possède un ancien système d'animations, qui ne passe pas par l'animator de Unity, ce qui m'empêchera par la suite de créer un fondu entre

les différentes animations du joueur. Lorsque le personnage est importé dans mon projet, je l'ai renommé en Player, adapté sa taille, et ensuite mis la caméra présente de base dans le projet juste derrière le personnage de manière à voir ce dernier à la 3ème personne. En effet c'est depuis cette caméra que nous verrons le jeu une fois lancé. Il faut donc bien l'adapter afin que nous voyons le joueur de dos comme dans un RPG classique. J'ai donc fait de la caméra un enfant de l'objet Player dans la hiérarchie pour que la caméra suive le joueur même si il se déplace. Ensuite j'ai fait glisser l'objet Player dans le dossier « Prefabs » dans l'inspecteur de fichiers Unity afin de créer une prefab du joueur. Une prefab correspond à un modèle que l'on sauvegarde afin de pouvoir le dupliquer où le réutiliser plus tard. Ici mettre le joueur en Prefab me permet de garder une sauvegarde du joueur au cas où les modifications que j'effectuerai dessus n'iraient pas. Il faut ensuite ajouter des composants à mon personnage. En effet, Unity possède beaucoup de composants de base que nous pouvons appliquer à des objets. Pour mon personnage j'ai appliqué un Rigidbody qui lui permettra de soumettre le personnage à la gravité. Il lui faut aussi un CapsuleCollider, qui nous permet de créer une zone de collision pour le joueur.

Il faut ensuite créer un script C# que l'on appliquera au joueur et qui nous permettra de gérer les déplacements du joueur et les animations notamment. Donc j'ai créé un script dans les dossier du jeu que j'ai nommé « CharacterMotor » et directement appliqué au personnage de la même manière que les deux précédents composants. Lorsque l'on créé un script depuis Unity, les bibliothèques et la syntaxe de base sont directement importés de base dans le script, ce qui facilite la création de scripts C#. Une fois dans le script, il nous faut créer une variable de type Animation qui nous permettra de stocker les différentes animations présentes de base dans le composant « Animation » du personnage. J'ai aussi créé 3 variables permettant de stocker la vitesse de déplacement en marchant, en courant et en tournant. Pour choisir quelles touches il faudra appuyer pour se déplacer dans les 4 directions possibles (avant, derrière, droite et gauche), j'ai créé 4 nouvelles variables de type « string » qui correspondent à des chaînes de caractères. Il me faut aussi une variable permettant le saut. Pour cela il faut créer une variable de type « Vector3 » qui correspondra a la force/vitesse sur l'axe correspondant (ici sur l'axe Y pour faire un saut à la verticale). Il me faut aussi créer une variable de type « CapsuleCollider » pour récupérer la capsule de collision du joueur qui nous servira plus tard. Pour remplir toutes les variables crées, cela se passe dans Unity. En effet l'inspecteur d'Unity nous permet de remplir et modifier les variables publiques directement pendant que nous prévisualisons le jeu dans l'inspector Game. Ce qui facilite grandement les choses pour faire des tests en temps réel.



Variables créées et remplies dans l'inspecteur de Unity

Une fois toutes les variables instanciées, il faut assigner les composants aux variables correspondantes. En effet pour les variables de type Animation et CapsuleCollider ne trouvent pas le composant correspondant tout seul. C'est donc dans la fonction *Start* que nous lions ces deux variables à leur composant dans le jeu, grâce à la commande *GameObject.GetComponent*<Nom du composant dans le jeu>*()*. Il faut absolument faire ces commandes dans la fonction *Start* car ces dernières doivent être appelées au lancement du jeu, sinon cela ne pourrait pas fonctionner.

Il faut ensuite ajouter les conditions pour que le personnage puisse bouger. Cela se passe dans la fonction *Update* qui se met à jour constamment pendant que le jeu est en exécution. Il faut donc détecter la touche que le joueur appuie sur son clavier pour exécuter l'action correspondante à la touche enfoncée si la touche possède une action. Heureusement, Unity possède une fonction pour cela, c'est la fonction *Input.GetKey(*Nom de la touche, q par exemple pour la touche « q »du clavier*)*. Une fois que le joueur appuie sur une touche correspondante à une action, que j'ai défini avec les 4 différentes variables de type « string », il faut exécuter l'action.

Pour le moment, je me suis concentré sur les mouvements du joueur. Donc si le joueur appuie sur la touche pour avancer, il doit avancer dans le jeu. Pour cela le personnage doit effectuer une translation sur l'axe Z pour avancer en avant. J'ai utilisé la fonction *transform.Translate(*mouvement sur l'axe x, sur l'axe y, sur l'axe z*)* avec pour paramètres 0 sur l'axe x et y car lorsque l'on appuie simplement sur la touche avancer, on se déplace simplement sur l'axe z, et *walkSpeed * Time.deltaTime* sur l'axe z qui correspond à la vitesse de déplacement en marchant que nous avons déjà instancié, multiplié par le temps. En effet, si dans le futur je créerai quelque chose qui ralentirait le temps, comme un sort par exemple, la vitesse de déplacement du personnage serait donc ralentie en même temps que la vitesse du monde. En multipliant donc la vitesse du joueur par le temps nous obtiendrons un effet d'accélération où de ralentissement en fonction de la vitesse a la quelle le temps passe dans le jeu.

Toujours dans la condition d'appui de la touche pour avancer, à la suite de la fonction de translation, il faut ajouter la fonction pour jouer l'animation de marche. Pour cela c'est très simple, il faut simplement utiliser la commande Unity *animations.Play(*« Nom de l'animation dans le composant Animation du personnage dans le jeu »*)*. En effet, étant donné que j'ai relié la variable *animations* au composant de type « Animation » dans Unity au préalable, la fonction peut directement trouver toutes les animations stockées dans le composant.

Pour la condition de sprint, c'est exactement la même chose sauf qu'il faut changer les variables correspondantes pour le nom de l'animation et pour la translation. Il faut également rajouter une fonction de condition *Input.GetKey(KeyCode.LeftShift)* ,qui permet de détecter si le joueur appuie sur la touche Left Shift sur son clavier. Ces deux conditions réunies permettent de pouvoir sprinter seulement si le joueur appuie sur la touche pour avancer et Left Shift en même temps. Seulement il faut donc modifier la condition pour marcher. En effet si je laisse comme ça, lorsque le joueur va appuyer sur l a touche pour avancer et Left Shift, il va sprinter, mais aussi marcher car la condition pour marcher est aussi satisfaite. Or l'animation de sprint et de marche sont différentes et cela va donc créer des soucis dans Unity. J'ai donc rajouté la condition *!Input.GetKey(KeyCode.LeftShift)* à la condition pour marcher, qui permet de détecter que la touche Left Shift n'est pas enfoncée.

Pour reculer, c'est encore exactement la même chose qu'avancer avec l'Input qui change aussi et qu'il n'y en a qu'un seul (car il ne peut pas y avoir de conflit étant donné que la touche appuyée est différente), sauf que la variable pour la translation est - *walkSpeed / 2* car lorsque l'on recule en

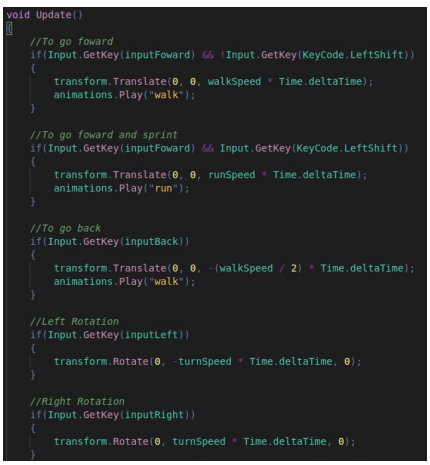
général nous allons moins vite que lorsque l'on avance, et la valeur est négative car on se déplace dans le sens inverse sur l'axe z pour pouvoir reculer.

Quant aux rotations, mêmes conditions que pour reculer avec la variable qui correspond à la rotation (droite ou gauche). Mais la fonction appelée est différente lorsque la condition est respectée. En effet, il faut utiliser *transform.Rotate(*Rotation sur l'axe x, sur l'axe y, sur l'axe z*)* et cette fois-ci le paramètre utilisé est sur l'axe y car je personnage effectue une rotation pour tourner sur soi même. Le paramètre est négatif pour tourner à gauche et positif pour tourner à droite.

Ensuite le personnage doit effectuer une animation quand il ne fait rien où quand il tourne sur place. Pour cela il faut simplement utiliser deux conditions, une qui vérifie que le joueur n'avance pas, et une pour vérifier qu'il ne recule pas non plus. Si ces conditions sont respectées alors l'animation « Idle », qui correspond à la respiration du personnage, peut se jouer.

Pour l'animation de saut c'est un peu plus compliqué. En effet, pour pouvoir sauter mon personnage doit être sur le sol, sinon il pourrait sauter à l'infini. Il faut donc vérifier qu'il soit en contact avec le sol avant de permettre de sauter. Pour cela j'ai créé un booléen *IsGrounded* qui utilise la fonction *Physics.CheckCapsule(playerCollider.bounds.center, new Vector3(playerCollieder.bounds.center.x, playerCollider.bounds.min.y -0.1f, playerCollider.bounds.center.z), 0.08f)*. Cette fonction renvoie vrai si il y a une collision avec la capsule de collision du joueur à 3 points différents. Au centre de la capsule, au centre de l'axe x et au centre de l'axe z pour avoir les deux extrémités de la capsule ainsi que le centre. En effet grâce à la détection de ces 3 points de collision, le joueur pourra sauter sur des pentes faibles, mais dès lors que le joueur se trouvera sur une pente plus raide, il ne pourra plus sauter car un des 3 points de collision ne touchera plus le sol.

Une fois le booléen de vérification *IsGrounded* créé, je peux donc l'utiliser en tant que condition pour sauter. Donc j'ajoute la condition d'Input pour vérifier que la touche de saut est enfoncée (j'ai d'ailleurs remarqué que la touche Espace correspond au KeyCode 0 et pas Space) ainsi que *IsGrounded*. Si les deux renvoient vrai on peut donc lancer le saut. Étant donné que mes scripts sont en C# je dois ajouter 2 lignes avant dans lancer le saut qui correspondent à une sorte de préparation au saut, uniquement nécessaire en C#. La première ligne consiste à stocker la vélocité du composant *RigidBody* de notre personnage, qui consiste à appliquer les effets de la gravité sur ce dernier. La deuxième ligne assigne la valeur de la vélocité sur l'axe y à la valeur du « Vector3 » *jumpSpeed* en y que j'ai instancié au début du script. Je peux donc permettre le saut après ces deux lignes en assignant *jumpSpeed* à la valeur de la vélocité du *RigidBody*.



```
void Update()
{
    //To go foward
    if(Input.GetKey(inputFoward) && !Input.GetKey(KeyCode.LeftShift))
    {
        transform.Translate(0, 0, walkSpeed * Time.deltaTime);
        animations.Play("walk");
    }

    //To go foward and sprint
    if(Input.GetKey(inputFoward) && Input.GetKey(KeyCode.LeftShift))
    {
        transform.Translate(0, 0, runSpeed * Time.deltaTime);
        animations.Play("run");
    }

    //To go back
    if(Input.GetKey(inputBack))
    {
        transform.Translate(0, 0, -(walkSpeed / 2) * Time.deltaTime);
        animations.Play("walk");
    }

    //Left Rotation
    if(Input.GetKey(inputLeft))
    {
        transform.Rotate(0, -turnSpeed * Time.deltaTime, 0);
    }

    //Right Rotation
    if(Input.GetKey(inputRight))
    {
        transform.Rotate(0, turnSpeed * Time.deltaTime, 0);
    }
```

```
//Stand animation
if(!Input.GetKey(inputFoward) && !Input.GetKey(inputBack))
{
    animations.Play("idle");
}

//Jump animation
if(Input.GetKeyDown(KeyCode.O) && IsGrounded()) //Space doesn't work in the editor
{
    //Jump preparation
   Vector3 v = gameObject.GetComponent<Rigidbody>().velocity;
    v.y = jumpSpeed.y;

    //Jump
    gameObject.GetComponent<Rigidbody>().velocity = jumpSpeed;
}
```

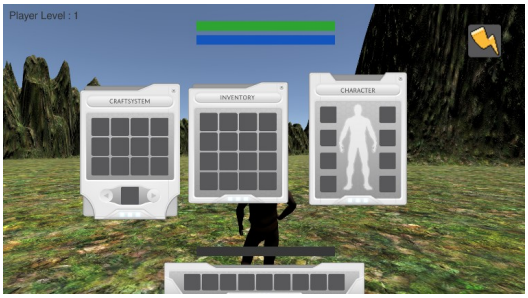
Code pour les différentes animations de déplacement, idle et saut

C) Création de l’inventaire et de la première interface du joueur

Lorsque j’avais un personnage jouable qui pouvait se déplacer, j’ai commencé l’inventaire. Pour commencer j’ai téléchargé l’asset Inventory Master sur l’Asset Store qui inclut un système d’inventaire complet avec de nombreux scripts que je peux réutiliser et l’interface. Une fois importé et rajouté dans la hiérarchie du projet, j’ai ajouté le composant *Player Inventory* à mon personnage. C’est un des scripts importés avec l’asset Inventory Master. Le script demande d’ajouter les différentes interfaces correspondantes dans l’inspecteur. En effet, parfois certains scripts vont demander de lier des objets existants de la scène/hiérarchie à des composants du script. Ici je dois par exemple lier *Inventory* avec le morceau de l’interface correspondant. Ou aussi *Character System* avec la fenêtre de l’inventaire qu’il faut. Dans le jeu, les touches pour ouvrir les différentes interfaces sont prédéfinies. La touche i permet d’ouvrir l’inventaire, k d’ouvrir l’interface de confection, et c pour ouvrir l’interface d’équipement du personnage.

Ensuite je dois modifier le script *Player Inventory* afin de l’adapter à mon projet. J’ai tout d’abord supprimé les différentes choses en rapport avec la barre de vie et de mana déjà présentes par défaut dans le script pour pouvoir créer mes propres barres de vie et de mana car celles par défaut ne correspondaient pas à ce que je voulais. Pour cela j’ai téléchargé un pixel blanc que j’ai importé dans le projet, puis changé en tant que *Sprite and UI*. Puis j’ai modifié l’interface d’inventaire que j’ai téléchargé en ajoutant une image que j’ai étiré pour donner la forme d’une barre de vie. C’est ici que le pixel m’a servi. En effet je l’ai appliqué à mon image/barre de vie en tant que texture et mis la couleur verte. Sans cela les couleurs n’apparaissaient pas. J’ai aussi renommé cette barre de vie en *CurrentHp* et je l’ai dupliquée. J’ai renommé la duplication en *BackgroundHp* et changé la couleur en rouge. J’ai ensuite lié les deux en créant un objet vide en le renommant *HealthBar*, et en faisant de *CurrentHp* et *BackgroundHp* ses enfants. Puis j’ai dupliqué *HealthBar* en adaptant la duplication à la barre de mana, comme par exemple changer la couleur verte en bleue, renommer les composants en *ManaBar*, *CurrentMana* et *BackgroundMana* et en changeant sa position.

Pour lier les deux barres au niveau de vie et mana de notre personnage, il faut retourner dans le script *Player Inventory*. J’ai donc commencé par rajouter deux variables de type « Image » *hpImage* et *manaImage* et je les ai initialisées dans la fonction *Start* à l’aide de la fonction *GameObject.find(*« Nom de la barre a bouger *Current*XX *»).GetComponent<Image>()*. Dans la fonction *Update*, j’ai créé une variable de type « float » qui va contenir la valeur du pourcentage de la vie actuelle, et remplir *hpImage* en fonction du pourcentage de vie actuelle ce qui changera la taille de *CurrentHp* dans le jeu et permettera de visualiser la vie restante en temps réel. Pour la barre de mana c’est exactement la même chose mais en adaptant les noms. Il faut ensuite changer les paramètres de type d’image de *CurrentHp* et *CurrentMana* en les mettant sur Filed Horizontal pour que l’image se remplissent correctement. Dans Unity, j’ai testé les variations des barres en changeant les valeurs directement dans l’inspecteur en jeu. Tout était fonctionnel.



Inventaire ouvert en jeu avec barre de vie et de mana

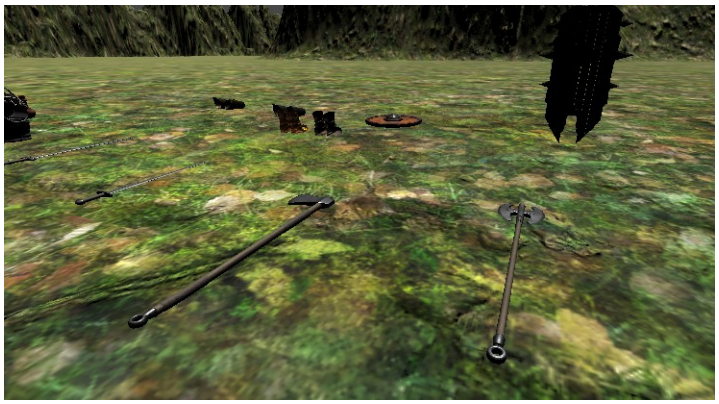
D) Création des objets à ramasser

Maintenant que j'ai l'inventaire, il me faut des objets à ramasser! Et mon asset d'inventaire possède beaucoup de choses pour gérer les objets à ramasser. Notamment une base de donnée qui permet de remplir beaucoup d'informations pour chaque objet. En effet, pour un seul objet il y a son nom, un ID unique, une description, un icône, un modèle 3D, sa valeur, son type d'objet, son poids, le maximum que l'on peut cumuler en une case d'inventaire, sa rareté et ses attributs.

Une fois la base de données trouvée, elle affiche beaucoup d'items par défaut. Je les ai donc tous enlevés afin de mettre mes propres items. J'ai également enlevé tous les items affichés dans l'inventaire de base. Il faut aussi que j'ajoute le tag *Player* à mon personnage pour que l'inventaire reconnaisse le personnage. J'ai donc commencé par ajouter deux épées et deux haches. Je leur ai ajouté le composant *PickUp Item* qui va permettre au joueur de récupérer ces 4 armes lorsqu'elles sont sur le sol, ainsi qu'un *RigidBody* pour la gravité et un *Box Collider* pour permettre les collisions, sinon le joueur ne pourra pas les ramasser.

Seulement j'ai remarqué que la distance pour récupérer un item au sol est très grande, je l'ai donc directement modifiée dans le script *PickUp Item*. Une fois tout cela effectué il faut ajouter les armes dans la base de donnée en remplissant les différentes informations de chaque arme sans oublier de changer l'ID pour chaque arme.

Ensuite il faut mettre l'ID de l'arme correspondante dans le composant *PickUp Item* dans l'inspecteur. Sinon le script ne saura pas de quelle arme il s'agit. J'ai également remarqué un autre problème. En effet, lorsque l'on équipe une arme, la fenêtre d'information de l'arme ne disparaît pas. J'ai réussi à trouver dans le script *ConsumeItem* comment régler ça en ajoutant une ligne *tooltip.desactivateTooltip()* qui enlève l'affichage de la fenêtre. C'est le même principe pour ajouter n'importe quel item dans le jeu. Il faut simplement faire attention au type d'item que l'on met dans la base de donnée, ses attributs et surtout son ID. En tout j'ai ajouté trois casques, deux plastrons, deux paires de chaussures, deux boucliers, une paires de gant, deux paires de gantelets et quatre armes. J'ai aussi créé des potions de vie et de mana, qui restaurent 20 points de vie ou de mana du joueur si il en consomme.



Capture d'écran de quelques items au sol

E) Création du système de dégâts et mort du joueur

Une fois avec de l'équipement tel que de l'armure et des armes, je vais commencer la partie dégâts et mort du personnage. Pour cela je dois faire des modifications dans le script *Player Inventory*. Dans les deux fonctions *OnGearItem*, qui permet d'ajouter les attributs de l'item équipé, et *OnUnEquipItem*, qui permet de retirer les attributs de l'item déséquipé, j'ai changé tous les préfixes « max » par « Current ». Dans le script *CharacterMotor* j'ai rajouté une variable de type « bool », qu'on assigne directement à faux, pour savoir si le personnage est mort ou non. Ce booléen va servir de condition pour pouvoir exécuter les mouvements ou actions dans la fonction *Update*. Si le personnage n'est pas mort il peut bouger et interagir, sinon il ne peut rien faire. De retour dans *Player Inventory*, j'ai rajouté une variable de type « CharacterMotor » et une autre de type « Animation » et je les initialise dans la fonction *Start* a l'aide de *GameObject.GetComponent*<Nom du composant dans le jeu>*()*.

Je créé ensuite une fonction *Dead()* qui lorsqu'on appelle cette fonction, met le booléen *IsDead* de *CharacterMotor* à vrai, et joue l'animation de mort du personnage. Puis j'ai créé une nouvelle procédure *ApplyDamage* qui prend un « float » en paramètre et qui correspondra au dégâts reçus. Dans cette procédure, on applique une équation à la vie actuelle qui consiste à soustraire le montant des dégâts reçus, moins la valeur de l'armure actuelle multipliée par la valeur des dégâts reçus, divisé par cent. Ce qui donne *CurrentHealth = CurrentHealth – (TheDamage – ((CurrentArmor * TheDamage) / 100))*. On ajoute également une condition qui consiste à executer la fonction *Dead()* lorsque la vie actuelle du personnage est inférieure ou égale à 0. Pour tester la prise de dégâts de mon personnage j'ai introduit une nouvelle touche qui, lorsque j'appuie dessus, exécute la procédure *ApplyDamage* avec 10 points de dégâts en paramètre.

F) Système d'attaque et affichage de l'équipement

Maintenant que le personnage peut avoir de l'équipement et prendre des dégâts, je vais me concentrer sur l'attaque du personnage et l'affichage de l'équipement équipé sur le personnage. Tout d'abord, j'ai modifié les textures du personnage car il possède déjà une armure de base, or cela ne me convenait pas. Les captures d'écran présentes dans le rapport ont été prises lorsque le projet fût terminé, je n'ai donc pas de représentation de l'ancien modèle du personnage disponnible. Par la suite, dans le script *CharacterMotor*, j'ai créé deux variables de type « float ». Une pour l'intervalle de temps durant laquelle on ne peut pas attaquer une deuxième fois, et une autre pour connaître le temps restant avant de pouvoir attaquer à nouveau. J'ai aussi rajouté un booléen qui permettra de savoir si le joueur est entrain d'attaquer ou non.

Donc j'ajoute une condition dépendante du booléen que je viens de créer. Si le joueur est entrain d'attaquer, alors le temps restant avant de pouvoir attaquer à nouveau diminue en fonction du temps. J'ajoute a la suite une autre condition, si ce temps restant est inférieur ou égal à 0, alors le temps restant devient égal a l'intervalle de temps durant laquelle on ne peut pas attaquer et le booléen *isAttacking* passe a faux. Ensuite j'ai créé une nouvelle fonction *Attack()* qui va passer *isAttacking* à vrai, et jouer l'animation d'attaque. Or il faut donc adapter les précédentes conditions d'animations car sinon il peut y avoir des conflits d'animations qui se jouent en même temps. C'est pourquoi j'ai ajouté la condition *isAttacking* doit être faux pour pouvoir lancer les animations de marche, de recul, et de sur place (Idle). Et je rajoute une nouvelle condition à la suite de la condition qui vient d'être rajoutée dans ces 3 cas, si l'on appuie sur le clic gauche de la souris, alors on

exécute la fonction *Attack()*. Dans Unity il ne faut pas oublier de remplir les valeurs nécessaires dans l'inspecteur sinon ça ne marche pas.

Ensuite il faut ajouter le rendu visuel des armes et des armures équipés sur le personnage. Car pour l'instant mon personnage peut s'équiper de tout ce qui est possible mais rien ne changera pour le visuel. Pour les armes il faut placer le modèle 3D dans la main du personnage en adaptant leur taille et leur rotation, pour en faire un enfant de cette main dans la hiérarchie. Comme ça, l'arme restera dans la main du personnage également pensant les animations. Pour les armures c'est quasiment la même chose sauf qu'il faut y fixer sur la partie du corps correspondante. Le casque sur la tête donc ce sera un enfant de la tête, le plastron sera lui un enfant du torse, les chaussures des pieds, etc. Pour chaque membre de l'équipement fixé, j'ai également désactivé le rendu visuel. En effet, ils s'activeront grâce à un script pour sélectionner lesquels afficher en fonction de l'équipement équipé. Je vais donc créer ce script qui aura le nom de *CheckItem* et qu'on appliquera à chaque emplacement d'équipement.

Dans ce script je vais créer une variable de type « int » pour stocker l'ID de l'item équipé, une variable de type « GameObject » pour stocker les membres du corps corespondant à l'item équipé, ainsi que la liste de toutes les items disponibles que je stocke dans une variable. Juste au dessus de la liste des items j'ai rajouté le mot clé [Serializefield] car sinon la liste n'apparaît pas dans l'inspecteur et nous ne pouvons pas la remplir avec les différents items possibles. Dans la fonction *Update* je vais vérifier en permanence si l'objet qui possède ce script à un enfant, donc si un item est équipé. Car quand aucun item n'est équipé, il n'y a donc aucun enfant dans les emplacement d'équipement. Dès lors que l'on équipe un item, l'emplacement d'équipement correspondant gagne alors un fils qui correspond à l'item en question. Donc si il y a un enfant, alors on stocke l'ID de l'item dans la variable d'ID créée au début du script. S'il n'y a aucun enfant, alors la variable contenant l'ID passe à 0 et je désactive l'affichage de l'item car si il il n'y a pas d'enfant, il n'y a pas d'item équipé donc rien ne doit être affiché.

A la suite de cette condition, j'ai ajouté une nouvelle condition qui regarde si l'ID stocké est égal à l'ID du premier item de la base de donnée et si il y a au moins un enfant. Si ces deux nouvelles conditions sont validées alors pour i est égal à l'ID de l'item correspondant dans la liste du script dans l'inspecteur, jusqu'au nombre d'items de la liste : si i est égal à l'ID de la liste de l'inspecteur alors on active le visuel de l'item possédant l'index i dans la liste des items.

Ensuite j'ai réitéré ces conditions pour chaque item du jeu, en changeant l'ID de l'item de la base de donnée et la valeur de i en fonction de l'item correspondant. Maintenant il faut infliger des dégâts en fonction de l'équipement du personnage. Pour cela, j'ai modifié le script *CharacterMotor*. Tout d'abord j'ai ajouté une variable de type « float » qui correspondra à la distance à laquelle on peut toucher des ennemis avec une attaque, et une variable de type « GameObject » qui correspondra à une ligne imaginaire dans le jeu. Ensuite j'avais oublié de modifier la fonction *Attack()* pour faire en sorte que le joueur ne puisse pas attaquer à l'infini sans avoir de temps de recharge entre les attaques. Donc il suffit juste de mettre l'animation d'attaque dans la condition qui dit que *isAttacking* est faux, donc que notre personnage n'est pas entrain d'attaquer. Il faut aussi ajouter une nouvelle variable de type « RaycastHit » que je nomme *hit*, ainsi que cette condition : if(Physics.Raycast(rayHit.transform.position, transform.TransformDirection(Vector3.forward), out hit, attackRange)) qui, si est validée, permettra de dessiner la ligne imaginaire si elle est en contact avec un objet ou une entitée, et donc d'infliger des dégâts à ce dernier.

Puis, de retour sur Unity, j'ai créé un nouvel objet que j'ai renommé *RayHit* que j'ai placé au niveau de l'extrémité extérieure du centre de la poitrine du personnage. Sans oublier de le mettre en enfant du joueur pour qu'il reste toujours collé au personnage. Dans *CharacterMotor*, j'ai initialisé dans la fonction *Start* la variable de type « GameObject » créée juste avant avec le *RayHit* créé dans Unity. Pour tester si l'attaque fonctionne, dans Unity, j'ai ajouté un rectangle que je frappais avec mon personnage, et j'affichais du texte dans la console pour voir si le jeu détectait bien une attaque provenant de mon personnage.

Personnage sans équipement d'équipé

Personnage avec l'équipement équipé et affiché

G) Création des ennemis et de leurs fonctionnalités

Je me suis donc ensuite attaqué à la création des ennemis. Pour commencer, j'ai généré un *NavMesh*. Ce dernier est un outil Unity, se trouvant dans les outils de navigation d'Unity, qui permet de définir les zones dans lesquels les ennemis peuvent se déplacer. Cet outil détecte automatiquement les zones possibles, mais on peut aussi définir les zones nous même et y modifier. Pour mon premier terrain, les zones détectées automatiquement étaient convenables, surtout pour effectuer des tests. Je n'ai donc pas eu besoin d'y modifier.

Ensuite, il faut télécharger un modèle 3D qui possède au moins 4 animation. Une animation de marche, une animation de sur place (Idle), une animation d'attaque et une animation de mort. J'ai donc trouvé un modèle gratuit sur l'asset store qui correspondait à mes critères. Une fois le modèle de l'ennemi importé, je lui ai ajouté le composant *Nav Mesh Agent* pour que l'ennemi connaisse les zones de déplacement possibles, et créé un nouveau script C# que j'ai nommé EnemyAI.

Dans ce nouveau script je vais faire tout ce qui est en rapport avec l'intelligence artificielle de l'ennemi et ses fonctionnalités. J'ai donc ajouté deux premières variables. Une de type « Transform » que j'appelle *Target*, qui correspondra à la cible que l'ennemi prendra en chasse si elle s'approche de trop près. Et une deuxième variable de type « UnityEngine.AI.NavMeshAgent » qui correspond à la zone sur laquelle l'ennemi peut se déplacer, que j'appelle *agent*. J'ai également initialisé la zone de déplacement dans la fonction *Start* en attribuant le composant correspondant dans Unity à l'aide de la fonction *GameObject.GetComponent*. Dans la fonction *Update*, j'ai assigné la destination du personnage à la position de la cible *Target*.

Grâce à cela, l'ennemi va suivre la cible. Seulement il l'a suivra en continu, et ce, peut importe la distance entre la cible et l'ennemi. J'ai donc ensuite ajouté le script à l'ennemi dans Unity et renseigné les différents paramètres afin de tester dans le jeu. Pour que l'ennemi se mette à suivre la cible seulement lorsque cette dernière se trouve dans une certaine zone autour de l'ennemi, mais aussi ajouter les animations et gérer l'attaque, il faut ajouter plusieurs variables. Six variables de type « float » dont une pour stocker la distance à laquelle l'ennemi va chasser la cible qu j'initialise directement à 10 (*chaseRange)*, une autre pour calculer la distance entre la cible et l'ennemi (*Distance*), une autre pour définir la distance à laquelle l'ennemi touche la cible lorsqu'elle attaque que je définis directement à 2,2 (*attackRange*), une autre pour stocker le temps durant lequel l'ennemi attend avant d'attaquer à nouveau (*attackRepeatTime*) que j'assigne directement à 1, une autre pour stocker le temps d'une attaque (*attackTime*), et une dernière pour stocker les dégâts infligés (*TheDamage*).

Pour récupérer les animations il faut également ajouter une variable de type « Animation ». Dans la fonction *Start* j'ai initialisé les animations de la même manière que la zone de déplacement et initialisé *attackTime* à *Time.time* pour qu'il soit égal au temps. Dans la fonction *Update*, j'ai supprimé l'assignation de la destination du personnage à la position de la cible *Target*, puis j'ai assigné *Target* à l'objet du jeu possédant le label « Player » avec la commande *GameObject.Find(« Player »).transform*.

A la suite j'ai calculé la distance entre la cible et l'ennemi pour que la distance soit constamment mise à jour en cas de déplacement d'un des deux personnages. En fonction de la distance, l'ennemi va effectuer diverses choses. En effet, si la distance entre les deux est supérieure à la distance de chasse de l'ennemi, alors le script va lancer la fonction *idle()*, qui consiste à lancer l'animation *Idle*.

Si la distance entre les deux est inférieure à la distance de chasse de l'ennemi et qu'elle est aussi supérieure à la distance d'attaque de l'ennemi, alors on lance la fonction *chase()* qui consiste à lancer l'animation de marche et assigner la destination de l'ennemi à la position de la cible *Target*. En dernière condition, si la distance entre les deux est inférieure à la distance d'attaque de l'ennemi, alorsle script lance la fonction *attack()* qui consiste à vérifier si le temps est supérieur à *attackTime*. Si c'est le cas, alors il lance l'animation *Hit*, inflige des dégâts à la cible à l'aide de la fonction *ApplyDamage* définie dans le script *PlayerInventory* que l'on récupère depuis la cible, et définit *attackTime* comme le temps ajouté à *attackRepeatTime*.

Pour que l'ennemi ne traverse pas le corps du joueur lors de l'attaque j'ai assigné *agent.destination* à *transform.position* car sinon l'ennemi se déplacera jusqu'au centre du personnage et le traversera donc. J'ai défini les 3 fonctions *idle(), chase(),* et *attack()* juste en dessous de la fonction *Update*. Lorsque j'ai testé le script sur Unity, j'ai remarqué que les animations de mon modèle 3D avaient un problème. En effet elles s'effectuaient mais elles n'avaient pas le bon rendu. J'ai donc tenté de cherché de nouveaux modèles 3D mais peu de modèles 3D possédant des animations sont disponibles sur l'Asset Store et internet. Et le peu que j'ai réussi à trouver avaient soit le même problème, soit ils utilisaient un modèle d'animation que je n'arrivais pas à adapter à mon script. J'ai donc abandonné les recherches de modèle 3D d'ennemis pour me concentrer sur la continuité de mon projet. Effectivement, le script ne posait aucun problème, le problème venait des animations en elle même.

Ensuite il faut permettre au joueur d'infliger des dégâts à l'ennemi. Pour se faire, j'ai ajouté un *Capsule Collider* à l'ennemi dans Unity pour qu'il puisse subir des collisions. Je lui ai aussi ajouté le label « Enemy » afin de le reconnaître dans les scripts. De retour dans le script *EnemyAi*, j'ai rajouté deux variables au début du script : *isDead* qui correspond à un booléen permettant de savoir si l'ennemi est mort ou non, et *enemyHealth* de type « float » qui correspond à la vie de l'ennemi. J'ai donc englobé tout ce qu'il y a dans la fonction *Update* dans une condition. Si l'ennemi n'est pas mort alors on exécute le précédent code, sinon il n peut plus rien faire. A la fin du script, j'ai créé une nouvelle fonction *ApplyDamage* qui va permettre d'appliquer les dégâts reçus à la vie de l'ennemi et si elle est inférieure à 0 alors on appelle la fonction *Dead()* qui consiste quant à elle à changer le booléen *isDead* à vrai et jouer l'animation de mort. Le principe est le même que pour le joueur, sauf que pour soustraire les dégâts reçus à la vie de l'ennemi, on ne gère pas la possibilité d'armure. Puis, je devais modifier le script *CharacterMotor*.

Tout en haut du script, j'ai ajouté une nouvelle variable *playerInv* de type « PlayerInventory » que j'ai initialisé dans la fonction *Start* de la même manière que les autres, afin de stocker la valeur des dégâts que le joueur inflige en fonction de l'arme équipée. Dans la fonction *Attack()*, j'ai ajouté une nouvelle condition dans la condition qui fait apparaître la ligne imaginaire, qui consiste à vérifier si le label de l'objet touché par la ligne imaginaire est égal à « Enemy ». Si c'est le cas alors on applique des dégâts à l'ennemi en récupérant la fonction *ApplyDamage* du script *EnemyAI* en lui passant *playerInv.currentDamage* en paramètres afin de lui infligé le montant des dégâts actuels du joueur.

En faisant des tests dans Unity, j'ai remarqué que le joueur jouait l'animation de mort à l'infini lorsqu'il mourrait. J'ai donc modifié la fonction *ApplyDamage* du script *PlayerInventory* en englobant le code dans la condition tant que *characterMotor.isDead* est faux.

J'ai ensuite décidé de rajouter une fonctionnalité qui permet à l'ennemi de revenir à son point de départ lorsque le joueur s'éloigne trop du joueur, et de rajouter une fonctionnalité de largage d'équipement aléatoire lorsque le joueur tue un enemi. Pour commencer, dans le script *EnemyAI*, j'ai rajouté deux variable, une variable *DistanceBase* pour stocker la distance qu'il y a entre l'ennemi et la base, et une variable *basePosition* de type « Vector3 » qui contiendra la position de la base. Dans la fonction *Start*, j'ai initialisé *basePosition* à *transform.position* qui correspond à la première position à laquelle l'ennemi est apparu lorsque le jeu à été lancé. Dans la fonction *Update*, j'ai calculé la distance entre la base et l'ennemi dans la variable *DistanceBase* de la même manière que la distance entre l'ennemi et le joueur.

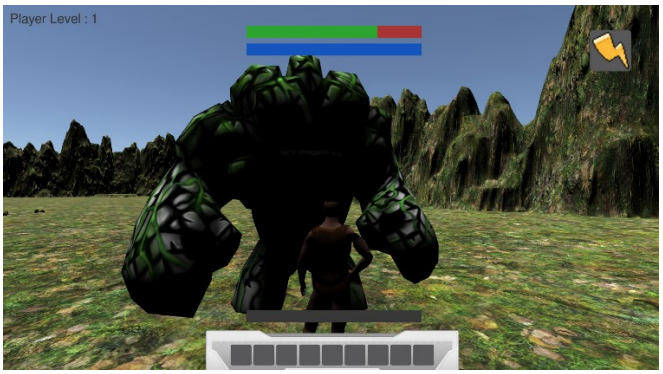
Il faut ensuite modifier la condition pour lancer la fonction *idle()* en ajoutant que *DistanceBase* doit être inférieur ou égal à 1. L'ajout de cette condition permet d'ajouter une marge pour que l'ennemi puisse effectuer l'animation *Idle*. En effet, parfois il ne pourra pas retourner à la distance exacte de sa première position. Puis j'ai ajouté une nouvelle condition qui vérifie si la distance entre les deux personnages est inférieure à *chaseRange* et que *DistanceBase* soit strictement supérieur à un. Si ces deux conditions sont vérifiées, alors on appelle la fonction *BackBase()*, que je définis en bas du script, qui consiste à jouer l'animation de marche et à assigner *agent.destination* à *basePosition* pour que l'ennemi retourne à sa position initiale.

Une fois le retour de l'ennemi à sa position initiale, je me suis occupé de la fonction de largage d'équipement aléatoire. Dans le script *EnemyAI*, j'ai ajouté une variable de type « GameObject[] » que j'ai nommé *loot* et qui correspond à une liste d'équipement que l'on pourra remplir dans Unity. Dans la fonction *Dead()*, j'ai ajouté une nouvelle variable aléatoire qui donne un nombre entier entre 0 et la longueur de la liste *loot*, et une variable de type « GameObject » *finalLoot* qui contiendra l'équipement correspondant à l'index de la liste *loot* correspondant au nombre aléatoire calculé juste avant. Ensuite je fais apparaître l'équipement à la position de mort de l'ennemi grâce à la fonction *Intsanciate(finalLoot, transform.position, transform.rotation)*.

Après avoir rempli la liste d'équipement possible dans Unity et testé l'apparition d'équipement à la mort de l'ennemi, j'ai remarqué que la boite de collision de l'ennemi était toujours active même lorsque l'ennemi est mort. Je l'ai donc désactivé dans la fonction *Dead()* avec la commande *GameObject.GetComponent<CapsuleCollider>().enabled* en le mettant à faux.

Modèle de l'ennemi à l'arrêt

Ennemi poursuivant le joueur (avec le problème des animations)



Ennemi infligeant des dégâts au joueur

H) Création du système de sort

Ensuite, je me suis concentré sur la création du système de sorts qui utiliserons le mana, présent dans mon projet depuis le début. Pour commencer, j'ai téléchargé un nouvel asset possédant un pack d'effets de particules pour créer un sort d'attaque et un sort de soin depuis l'Asset Store. Après avoir importé le pack dans mon projet, j'ai mis la particule qui correspondrait le plus à une particule de projectile électrique dans ma scène.

Dans Unity les particules animées utilisent un système spécifique s'appelant *Particule System*. C'est depuis ce composant que l'on peut modifier les paramètres spécifiques aux particules comme la vitesse d'animation ou la durée d'apparition de la particule par exemple. J'ai ensuite modifié un paramètre du *Particule System* en mettant le paramètre *Simulation Space* en *local* et non en *world* pour permettre de déplacer la particule correctement, car sinon je pouvais déplacer les composants de la particule, mais pas le rendu visuel.

J'ai aussi augmenté la vitesse de l'animation car j'ai remarqué que lorsque je lance l'animation de la particule, il y a un certain temps qui s'écoule entre le moment où l'animation se lance, et où la particule apparaît. Et lorsque j'accélère la vitesse d'animation, cela réduit ce délai d'apparition. Il faut aussi lui ajouter un composant *Rigidbody* et un *Sphere Collider* afin de pouvoir détecter les collisions. Pour le sort de soin, c'est exactement la même chose mais avec la particule qui correspond le mieux à un sort de soin sans ajouter le *RigidBody* et le *Sphere Collider*. Ensuite dans

le script *CharacterMotor*, j'ai ajouté la bibliothèque *UnityEngine.UI* pour pouvoir utiliser des objets de l'UI d'Unity, puis j'ai ajouté de nouvelles variables au début du script. Quatre variables de type « GameObject », une pour stocker l'objet Unity du sort de foudre (*lightningSpellGO*), une pour l'objet du sort de soin (*healSpellGO*), une autre pour stocker un nouveau *RayCast* pour les sorts (*raySpell*) afin de ne pas utiliser le *RayCast* des attaques au corps à corps, et une dernière pour stocker l'image du sort équipé (*spellHolderIMG*).
J'ai ensuite ajouté quatre variables de type « float ». Une pour le coût en mana du sort de foudre (*lightningSpellCost*), une pour le coût du sort de soin (*healSpellCost*), une pour la vitesse de déplacement du sort de foudre (*lightningSpellSpeed*) et une pour stocker le montant de vie que le sort de vie redonne au joueur (*healSpellAmount*).

Il faut également ajouter un identifiant unique pour chaque sort (*lightningSpellID* et *healSpellID*), une variable pour stocker l'ID du sort selectionné (*currentSpell*) que j'assigne directement à 1, une pour stocker le nombre total de sorts (*totalSpell*), et un objet de type « Sprite » pour stocker les images qui serviront a voir quel sort est actuellement équipé. Puis, sur Unity, j'ai ajouté l'interface graphique des sorts à l'interface globale de mon jeu. Pour cela j'ai simplement copié le rendu graphique d'un emplacement de l'inventaire, puis rajouté une deuxième image par dessus qui contiendra le logo du sort équipé, en faisant de cet ajout un enfant de l'inventaire et en le renommant « SpellHolderIMG ».

J'ai également ajouté le *RayCast* des sorts en déhors de la zone de collision du personnage, afin d'éviter le contact du sort de foudre avec le joueur lorsqu'il l'utilise. Dans le script *CharacterMotor*, j'ai initialisé *spellHolderIMG* dans la fonction *Start* avec *GameObject.Find(« SpellHolderIMG »)* ainsi que *raySpell* de la même manière. Ensuite dans la fonction *Update*, j'ai ajouté les conditions de changement de sort. Si Unity détecte que l'on utilise la molette de la souris en la faisant rouler vers l'utilisateur, avec la méthode *Input.GetAxis(« Mouse ScrollWheel ») < 0*, alors on effectue une nouvelle vérification. Si *currentSpell* est inférieur ou égal à *totalSpell* et que *currentSpell* est diférent de 1, alors *currentSpell* diminue et permet de changer l'ID du sort sélectionné.

J'ai refait la même chose pour lorsque le joueur fait rouler la molette de la souris en face de lui, en modifiant la deuxième vérification pour que *currentSpell* soit supérieur ou égal à 0, et qu'il soit différent de *totalSpell*, car sinon on ne peut pas augmenter l'ID étant donné qu'on est déjà au dernier sort. A la suite de ces nouvelles conditions, j'ai ajouté les conditions de changement d'image en fonction du sort équipé. Si *currentSpell* est égal à *lightningSpellID* alors on initialise *spellHolderIMG* avec l'image du sort de foudre. Et si *currentSpell* est égal à *healSpellID* alors on initialise *spellHolderIMG* avec l'image du sort de soin.

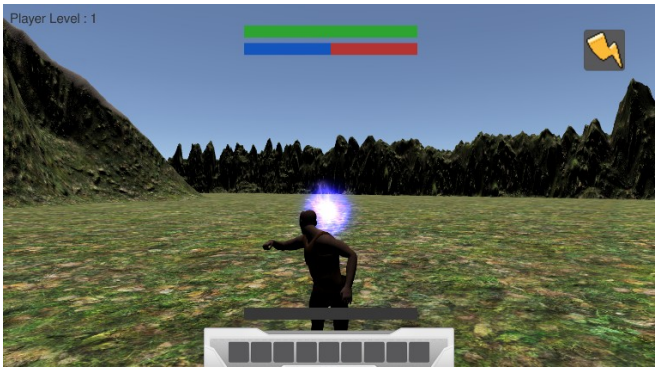
Puis, à la fin du script, j'ai créé une nouvelle fonction *Spell()* qui va consister à envoyer le projectile de foudre et faire apparaître le sort de soin. Dans cette fonction, on va ajouter une condition pour chaque sort. Si *currentSpell* est égal à *lightningSpellID*, que le joueur n'est pas entrain d'attaquer et que le mana disponible du joueur est supérieur ou égal à *lighningSpellCost*, alors on joue l'animation d'attaque, puis on instancie un « GameObject » qui correspondra à la particule de foudre en lui ajoutant une force, en soustrayant le coût du sort de foudre au mana du joueur et en mettant le booléen *isAttacking* à vrai.

Pour le sort de soin la condition est la même chose mais avec les variables du sort de soin et avec une condition en plus qui consiste à vérifier que la vie actuelle du joueur soit en dessous de la vie

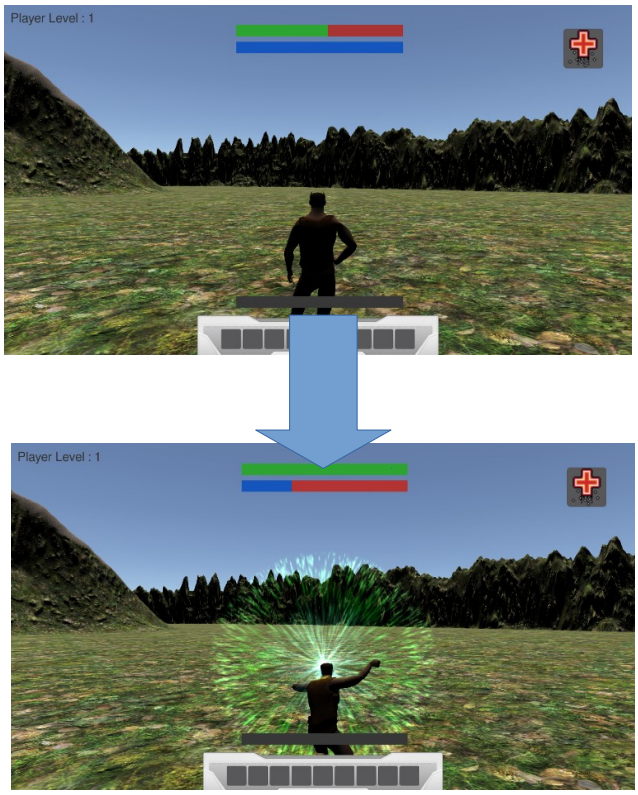
maximale. En effet, avec cette condition le joueur ne pourra pas gaspiller du mana en utilisant le sort de soin alors que ça vie est déjà pleine.

Une fois que les conditions du sort de soin sont validées, alors on joue l'animation d'attaque, on instancie directement l'animation de la particule, on soustrait le coût du sort de soin au mana du joueur, on ajoute le montant du soin ajouté par le sort à la vie du joueur et on met le booléen *isAttacking* à vrai. Pour faire en sorte que le sort de foudre disparaîsse au bout d'un certain temps sans rien toucher, et qu'il inflige des dégâts à un ennemi, j'ai créé un nouveau script *spellCollision*. Dans ce script j'ai créé une variable float *spellDamage* qui correspond aux dégâts infligés par le sort de foudre.

J'ai également supprimé la fonction *Update* car elle est inutile pour ce script. Dans la fonction *Start*, j'ai utilisé la fonction *Destroy(gameObject, 5)* pour détruire la particule lorsqu'elle ne touche rien au bout de 5 secondes. A la suite de la fonction *Start* j'ai créé une fonction *OnCollisionEnter* qui prend en paramètre une collision. Dans cette fonction je vérifie si le tag de la collision est égal à « Enemy ». Si c'est le cas, alors on applique les dégâts du sort à l'ennemi, et si la collision ne possède pas le tag « Player », alors on détruit la particule. Pour faire disparaître le sort de soin, j'ai créé un autre script *Auto Destruct* qui consiste a simplement instancier une variable *timer* de type « float » et utiliser la fonction *Destroy(gameobject, timer)* dans la fonction *Start*. Cela va détruire l'entité au bout du temps choisi dans Unity grâce à la variable *timer*. J'ai ensuite pensé que si le joueur possède 99 % de vie et qu'il utilise le sort de soin, sa vie va dépasser la valeur maximale possible. J'ai donc rajouté dans le script *CharacterMotor* la condition consistant à vérifier si la vie actuelle est supérieure à la vie maximale, alors la vie actuelle devient égale à la vie maximale.



Utilisation du sort de foudre



Utilisation du sort de soin

I) Création du système d'expérience, de compétences et adaptation des composants de l'interface

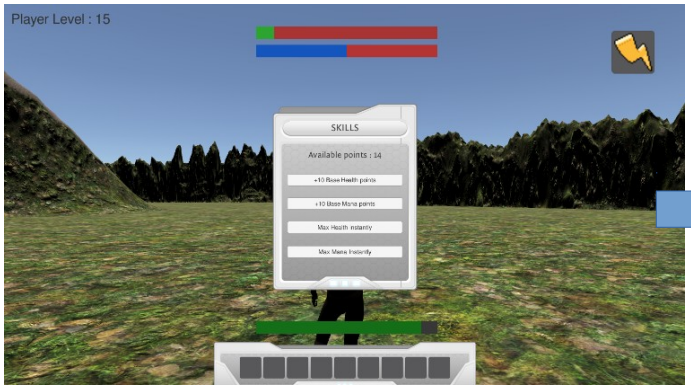
Dans un jeu de style RPG, le joueur possède un niveau qu'il peut augmenter en tuant des ennemis par exemple, qui lui permettent d'obtenir diverses améliorations. J'ai donc décidé d'inclure un système de niveau et d'expérience à mon jeu. Pour commencer, dans le script *Player Inventory*, j'ai créé six nouvelles variables. Une variable de type « Image » afin de stocker l'image de la progression de la barre d'expérience, une de type « Text » pour stocker l'affichage du niveau actuel du joueur, un de type « int » pour stocker le niveau du joueur que j'initialise directement à 1, et enfin trois de type « float » dont une pour l'expérience actuelle du joueur que j'initialise a 0, une pour le maximum d'expérience nécessaire pour passer un niveau que j'initialise à 100, et une pour stocker un nombre nécessaire pour multiplier le nombre d'expérience nécessaire pour passer au niveau suivant à chaque passage de niveau (*rateXP*). En effet, cela va permettre d'augmenter l'expérience nécessaire pour chaque niveau et rajouter un peu de difficulté.

De retour sur Unity, il faut maintenant ajouter les différentes représentations graphiques du niveau et de l'expérience. Pour se faire, j'ai copié la barre de vie de l'inventaire, et je l'ai placée en bas de l'écran. J'ai ensuite changé les couleurs et je l'ai renommée en *experienceBar*. Pour l'affichage du niveau actuel il faut simplement rajouter un texte, que j'ai placé en haut à gauche de l'écran, en changeant le texte pour « Player Level : 1 » qui sera mis à jour grâce au script.Dans le script *Player Inventory*, j'ai initialisé la barre d'expérience ainsi que le texte du niveau du joueur dans la fonction *Start* de la même manière que précédemment en récupérant l'objet dans Unity. Dans la fonction *Update*, j'ai rajouté une condition pour le passage de niveau. Si l'expérience actuelle du joueur est supérieure ou égale à l'expérience maximale pour passer un niveau, alors on créé une variable « float » qui contiendra le reste de l'expérience gagnée si le joueur gagne plus d'expérience qu'il est nécessaire d'avoir pour passer un niveau, on incrémente le niveau du joueur de 1, on change le texte du niveau en le mettant à jour avec le niveau actuel, on change la valeur de l'expérience actuelle en 0 ajouté au reste d'expérience pour ne pas perdre l'expérience gagnée, et enfin on multiplie l'expérience maximale par *rateXP*.

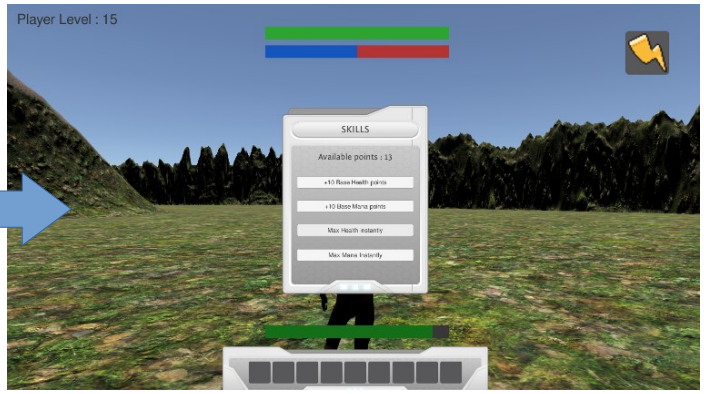
Pour l'affichage de la progression de la barre d'expérience, c'est exactement la même chose que pour la barre de vie mais en adaptant avec les variables de l'expérience. Pour faire en sorte que le joueur gagne de l'expérience lorsqu'il tue un ennemi, j'ai simplement ajouté une incrémentation de l'expérience actuelle du joueur dans la fonction *Dead()* se trouvant dans le script *EnemyAI*. Ensuite il fallait ajouter un nouveau panneau de l'inventaire afin de pouvoir créer la possibilité de dépenser des points de compétences. Dans Unity, j'ai ajouté une nouvel image en lui mettant la même texture que la fenêtre d'inventaire, en rajoutant un texte « Skills » en titre, ainsi qu'un texte « Available points : 0 » que je mettrai grâce à un script. J'ai également ajouté quatre boutons, un pour ajouter 10 points de vie à la vie maximale possible, un pour ajouter 10 points de mana au mana maximal du joueur, un pour remettre la vie du joueur à sa valeur maximale, et un dernier pour remettre le mana à sa valeur maximale. J'ai ensuite créé un nouveau script *playerSkills* dans lequel j'ai importé la bibliothèque *UnituEngine.UI*. Puis j'ai créé de nouvelles variables dont une pour stocker la fenêtre de compétences précédemment créée, une pour stocker le texte des points disponibles de la fenêtre de compétences, une pour stocker les points disponibles, une pour stocker la touche nécessaire pour ouvrir la fenêtre des compétences, une pour stocker un booléen permettant de voir si la fenêtre est ouverte ou non, et une pour stocker l'inventaire du joueur. Dans la fonction *Start*, j'ai initialisé l'inventaire du joueur en récupérant l'objet dans Unity.

Dans la fonction *Update*, je vérifie si on appuie sur la touche pour ouvrir la fenêtre, si c'est le cas alors on inverse le booléen de vérification d'ouverture de la fenêtre, cela permet d'ouvrir et de fermer la fenêtre avec la même touche. Ensuite, si le booléen de vérification d'ouverture de la fenêtre est égal à vrai, alors je change le texte en le mettant à jour avec la variable contenant les points disponibles et j'active la fenêtre des compétences, sinon je désactive la fenêtre. A la suite, il faut créer une fonction pour chaque bouton de la fenêtre. La première *addHealthMax* qui prend en paramètre le montant de la vie a ajouter, qui consiste a vérifier qu'il y a au moins 1 point disponible, et si cela est vérifié, alors on ajoute la valeur passée en paramètre de la fonction à la vie maximale et à la vie actuelle, et on soustrait 1 au nombre de points disponibles. Pour *addManaMax*, qui permet d'augmenter la valeur maximale du mana possible, c'est exactement la même chose que pour la vie, mais avec les variables de mana. Ensuite *LifeMax()*, qui permet de mettre la vie actuelle du joueur à sa valeur maximale, on vérifie qu'il y ait au moins 1 point disponible, et si c'est le cas, alors on change la valeur de la vie actuelle du joueur pour lui mettre la valeur de la vie maximale, et on enlève un point de compétence.

Pour *ManaMax()*, qui permet de mettre le mana actuel du joueur à sa valeur maximale, c'est la même chose que *LifeMax()* mais avec les variables de mana. Ensuite pour ajouter le gain de point de compétence au passage d'un niveau, il faut modifier le script *Player Inventory* en ajoutant une variable de type « playerSkills » au début du script et l'initialiser dans la fonction *Start*. Ensuite dans la vérification d'expérience pour passer un niveau, il faut rajouter un ligne qui permet d'ajouter 1 au nombres de points disponibles de *playerSkills*. En testant le système d'expérience sous Unity, j'ai remarqué que les éléments d'interface ne s'adaptent pas vraiment en fonction de la taille de l'écran. J'ai donc décidé de régler ce petit problème. En effet, la résolution de ce problème est plutôt simple, il faut ajouter un composant de type *Rect Transform* aux éléments de l'UI afin de pouvoir fixer les éléments à un endroit dans l'écran. Mais tout d'abord il faut ajouter le composant *Canvas Scaler* à l'inventaire (qui constitue tout l'interface/UI). J'ai donc changé le paramètre de ce composant *UI Scale Mode* en *Scale With Screen Size* et en mettant la résolution de référence à 1600 x 900 pixels, qui correspond à la résolution de mon écran, afin de changer la taille des composants en fonction de l'écran. Ensuite, pour ajouter les composants *Rect Transform* j'ai dû ajouter un image au différents morceaux de l'interface que je voulais fixer. En effet, une image est considérée comme un élément de UI et permet donc d'ajouter automatiquement un *Rect Transform*, que nous ne pouvons pas ajouter directement. Pour chaque image ajoutée, il faut fixer le *Rect Transform* dans l'endroit souhaité. Une fois tout cela terminé, je changeais la résolution de l'inspecteur dans Unity, afin de voir si l'interface s'adaptait bien à l'écran.



Joueur niveau 15 avec 14 points de compétences



Utilisation d'un point de compétence pour remettre la vie au niveau maximal

V – Retour sur expérience

Durant la création de ce projet sous Unity 3D, j'ai appris énormément de nouvelles choses dans le domaine du développement C# et l'utilisation du moteur Unity. En effet, j'ai appris comment utiliser des modèles 3D pour leur appliquer des fonctionnalités, comme les boîtes de collisions, où des scripts permettant d'utiliser des composants déjà présents dans le modèle 3D tels que les animations. J'ai également appris à créer des scripts entièrement pour créer des fonctionnalités telles que les déplacements du joueur ou encore créer une intelligence artificielle permettant aux ennemis de se déplacer dans une zone définie, attaquer le joueur ou une entité si l'entité en question se trouve dans un certain périmètre autour de lui, et lui permettre de revenir à sa position de départ si l'entité est trop loin de l'ennemi. J'ai aussi pu apprendre comment réutiliser des scripts et des assets d'ores et déjà créés, et à les adapter à mon projet,

La méthode de travail que j'ai utilisée tout au long du projet, consistait à travailler totalement en autonomie, et de chercher de l'aide sur internet comme des forums ou des vidéos YouTube. Cette méthode de travail est pour moi ce qui se rapproche le plus d'une situation réelle en entreprise, car parfois, personne ne sera en mesure de m'aider lorsque je rencontrerai des difficultés. Je devrais donc me débrouiller par moi même.

VI – Conclusion

En conclusion, j'ai créé un projet avec un terrain en 3D, un personnage jouable pouvant se déplacer, interagir avec des items, attaquer, mourir et utiliser des sorts, un système d'inventaire et d'équipement complet avec des statistiques, attributs et rareté spécifique à chaque item. J'ai également créé un système de pouvoirs, un système d'expérience et de compétences, une intelligence artificielle pour les ennemis leur permettant de se déplacer, suivre le joueur, attaquer le joueur, revenir a leur position de départ si le joueur s'échappe, et lâcher des items aléatoires lorsqu'ils meurent.
Hélas, je n'ai pas eu le temps de créer un système de quêtes jouables par le joueur. Je n'ai également pas réussi à trouver un modèle 3D gratuit avec des animations qui fonctionnent entièrement d'ennemi pour mon projet.

Par rapport à ma formation, ce projet m'a permis de renforcer mes compétences de programmation C#, mais aussi d'apprendre beaucoup de nouvelles choses et techniques, notamment pour l'adaptation de l'interface où il faut ajouter un *Rect Transform* a l'objet afin de le fixer dans un endroit de l'écran. Je me suis également amélioré de ma recherche de solutions sur internet, optimisant mon temps.

Ce projet m'a également permis de voir que je porte un réel intérêt à la programmation C#, mais aussi à la création de jeux vidéos. En effet, la réalisation de ce projet m'a beaucoup plu, notamment grâce aux possibilités de personnalisations infinies qui m'ont permis de réaliser tout ce que je voulais, ou presque.

VII – Annexes

Annexe 1 : Script CharacterMotor.cs

https://forge.univ-lyon1.fr/p1805132/rpg-stage-bonis-alexis/-/blob/master/CharacterMotor.cs

Annexe 2 : Script ennemyAI.cs

https://forge.univ-lyon1.fr/p1805132/rpg-stage-bonis-alexis/-/blob/master/ennemyAI.cs

Annexe 3 : Script autoDestruction.cs

https://forge.univ-lyon1.fr/p1805132/rpg-stage-bonis-alexis/-/blob/master/autoDestruction.cs

Annexe 4 : Script checkItem.cs

https://forge.univ-lyon1.fr/p1805132/rpg-stage-bonis-alexis/-/blob/master/checkItem.cs

Annexe 5 : Script PlayerInventory.cs

https://forge.univ-lyon1.fr/p1805132/rpg-stage-bonis-alexis/-/blob/master/PlayerInventory.cs

Annexe 6 : Script playerSkills.cs

https://forge.univ-lyon1.fr/p1805132/rpg-stage-bonis-alexis/-/blob/master/playerSkills.cs

Annexe 7 : Script spellCollision.cs

https://forge.univ-lyon1.fr/p1805132/rpg-stage-bonis-alexis/-/blob/master/spellCollision.cs