Compte-rendu du TP de Jeu vidéo sous Unity

LAMBERT Sylvain - PETTEX Matthieu - WYBRECHT Lucas

Rendu le 4 janvier 2023

Table des matières

1 Introduction

Unity est un moteur de jeu contenant de nombreuses solutions pour créer des représentations graphiques. On peut donc s'en servir pour créer un jeu vidéo répondant à plusieurs critères :

— Un univers permettant l'immersion du joueur ;

— Un gameplay, toutes les possibilités d'interaction que possède le joueur ;

— Un objectif : ce qui veut dire un début et une fin au jeu une fois l'objectif atteint.

Ainsi, on a ici les conditions les plus simples de création d'un jeu vidéo.

2 Genre du jeu

La première réflexion à avoir est le type de jeu qu'on cherche à mettre en place.
Dans le cadre de ce TP, il s'agit d'un jeu de type RPG (très simple) où l'on contrôle un personnage ayant pour objectif de ramasser un objet (un casque) qui n'apparaît qu'après avoir vaincu tous les ennemis de la zone.

3 Création du terrain

La première partie concerne la création du terrain. Ici, on va juste importer un asset déjà créé qui prend la forme d'un grand parc naturel. Le seul travail effectué est donc l'importation des assets et l'utilisation du *component* de *Box Collider* afin de gérer les collisions avec les autres *GameObjects* de la scène.

4 Création du joueur

Ensuite, il faut le personnage que l'on incarne. On choisit de plus de jouer à la $3^{ème}$ personne.
Pour suivre constamment le personnage avec la caméra, on lie cette dernière au personnage en tant qu'enfant du joueur. Ainsi, dès que le personnage se déplace, la caméra se met à le suivre. On crée une *Prefab* du personnage afin de sauvegarder tous les ajouts effectués par rapport à l'asset de base. Ensuite, on lui ajoute un *RigidBody* afin que le personnage soit soumis à la gravité. On "freeze" également les rotations pour éviter un bug qui fait parfois tomber le personnage de la surface.

Pour gérer les collisions, on ajoute également un *CapsuleCollider*. On adapte juste le taille ensuite afin d'être le plus réaliste possible.

On crée ensuite le script *CharacterMotor* pour faire bouger le personnage et l'animer. Il faut donc créer différentes variables :

— Une variable symbolisant l'*Animator* qui va gérer les animations.

— Des variables de vitesse pour le personnage (par exemple une pour marcher la deuxième pour courir). On les déclare en public afin de pouvoir les modifier immédiatement depuis l'interface.

— Des variables pour détecter le sens de déplacement.

— Un vecteur 3D donnant la direction du saut

— Un *component CapsuleCollider*

Après, dans la fonction *Start*, on référence la variable stockant les animations en lui donnant le composant *Animator* du personnage. On référence également le *CapsuleCollider* en lui appliquant celle donnée au personnage.

Pour gérer les différentes transitions d'animations de l'*Animator* lors de l'appui sur les touches, on fait juste en sorte de jouer avec des conditions selon la touche utilisée et de mettre à *true* ou *false* des paramètres booléens pour lancer ensuite l'animation souhaitée.

5 Ajout de l'inventaire

Un personnage est toujours lié dans un RPG à un inventaire afin de visualiser un objet qui a été récupéré au sol par exemple. Pour ce faire, on importe un asset qui contient d'office une interface d'inventaire. Toute l'interface est sous forme de canvas. Il faut ensuite lier l'élément *PlayerInventory* compris dans l'asset. Cela permet d'obtenir l'inventaire et l'équipement lié au personnage. On ajoute à ce canvas une barre de vie pour afficher constamment à un endroit de l'écran la santé actuelle du personnage. C'est juste deux textures blanches que l'on recolore afin de choisir la couleur de fond de la santé (maximale) et la couleur en premier-plan (santé actuelle).

On crée en plus dans le script *Player Inventory* une portion de code permettant de gérer la barre de vie. On récupère les composants de santé créés dans l'interface tout d'abord dans la fonction de démarrage (*Start*). On calcule ensuite le rapport entre la vie actuelle et la vie maximale dans le script dans la fonction de mise à jour (*Update*).

6 Ajout des items

Dans cette partie, on gère l'apparition des items dans la scène. Les items sont par exemple les équipements que l'on peut donner au personnage afin de l'améliorer. On prend donc un asset d'arme pour tester le fonctionnement. On choisit dans le cadre de ce jeu une épée. On garde comme composantes de l'objet seulement le *Mesh Filter*, le *Mesh Renderer* et le *Material*. Ensuite, on ajoute un script du nom de *PickUpItem*. Ensuite, comme d'habitude, on ajoute un *Rigidbody* ainsi qu'un *Box Collider*. On ajoute après l'arme fabriquée dans le dossier *Prefabs*. Il faut ensuite modifier les champs dans la base de données *ItemDatabase* afin que l'objet affiché dans l'inventaire corresponde à l'arme choisie.

7 Dégâts subis et mort du personnage

Il faut savoir gérer la santé du personnage selon les différentes actions dans le jeu. Le principe est simple : le personnage perd de la santé quand il prend un coup et meurt quand la santé tombe à zéro. Ici on crée dans le script *Player Inventory* une fonction d'application des dégâts prenant en paramètre les dégâts qui fonctionne selon l'équation suivante, tout en prenant soin de donner au script les composantes d'animation et de mouvement du personnage :

$$Health = Health - (Dégâts - (Armure \times Dégâts)/100)$$

De plus, si l'on finit avec des points de vie égaux ou inférieurs à 0, on appelle la fonction *Dead* qui active l'animation de mort et empêche le mouvement du personnage.

On crée dans le script *CharacterMotor* le booléen *IsDead* qu'on met en public pour que l'interface ait accès à l'information. Ensuite on modifie la fonction de déplacement en ajoutant une condition sur la survie du personnage, il ne peut logiquement pas bouger s'il est mort. Un premier test possible est la vérification de la perte de santé à l'aide d'une touche permettant d'infliger directement des dégâts au personnage pour voir si la mort est bien effectuée.

8 Attaque du personnage et affichage de l'arme

Ici, on gère l'attaque du personnage et les dégâts de son arme. On modifie le script *CharacterMotor* et on crée le script *checkWeapon*. Pour le premier script , on crée deux variables *private* donnant le cooldown actuel ainsi qu'un booléen signalant une phase d'attaque ou non. Ensuite, on crée une variable publique pour fixer le temps de cooldown, et une autre donnant la portée d'attaque du personnage. On crée ensuite une fonction d'attaque qui passe le booléen d'attaque à *true* et lance une animation uniquement si le personnage n'est pas déjà en train d'attaquer. Ensuite on crée un *RayCastHit* pour voir si on touche bien une cible lors de l'attaque.

Aussi, on crée à la fin de la boucle de mise à jour quelques lignes pour que si le personnage attaque, son temps de cooldown est réduit. De plus, si le cooldown est inférieur à 0, le booléen d'attaque repasse à *false*. On doit donc penser à modifier les conditions d'animation dans le script pour ne pas interrompre l'attaque si on veut se déplacer.

On place ensuite l'épée dans la main du personnage en la plaçant comme enfant de la main du personnage.

On crée ensuite le second script énoncé plus tôt qui vérifie le port d'arme du personnage. On récupère l'ID de l'arme actuelle dans une variable privée. On crée une liste publique qui stocke toutes les armes également. Dans la mise à jour, on vérifie régulièrement si une arme est ajoutée à l'emplacement d'équipement du personnage. On stocke l'ID de l'équipement ajouté ensuite. On lie ensuite l'arme dans la main du personnage à une arme de la base de données.

9 Ajout des ennemis

Les ennemis sont une partie importante du jeu étant donné qu'ils sont nécessaires à la fin du jeu. Les assets utilisés pour les définir sont importés. On va agir au niveau de leur comportement afin qu'ils attaquent dès que le personnage joué se situe à une certaine distance d'eux.

Tout d'abord, il faut délimiter le territoire de déplacement des ennemis. Pour ce faire, on utilise le sous-menu *Navigation* du menu *Window* d'Unity qui possède des outils pour sélectionner les zones où l'ennemi peut se déplacer. Cela crée ainsi un *NavMesh*.

On importe ensuite les ennemis en leut ajoutant en *component Nav Mesh Agent* afin que leur déplacement soit limité.

10 Conclusion

Finalement, on a développé le type de jeu souhaité au début. On a réussi à créer un jeu correspondant aux 3 critères qui étaient fixés :

— L'univers naturel qui donne toute la cohérence aux personnages présents dans le jeu.

— Le gameplay de type RPG, où l'on a implémenté plusieurs fonctionnalités, avec une barre de santé, des ennemis à affronter, des *loots* présents pour s'équiper.

— Un objectif très simple qui est la collecte d'un objet après avoir battu tous les ennemis présents dans la zone.

11 Références

— Lien vers l'Asset Store de Unity : Asset Store

— Tutoriel sur lequel s'inspire le jeu : Créer un RPG facilement sur Unity

— Tutoriel pour savoir comment utiliser les transitions d'animations de l'*Animator* : How to use Animation Transitions (Unity Tutorial)

— Tutoriel pour savoir comment relancer le jeu/la scène à l'aide d'un bouton : Unity - How to Reset / Reload Scene with C# | Easy Tutorial