Classe Passerelle 2019-2020

17/06/2020

KEY SEEKER

Rapport de projet | Florent Didion & Thibault Chatillon

1. Généralités

1. Unity

Unity est un moteur de jeu multiplateforme (smartphone, ordinateur, consoles de jeux vidéo et Web) utilisant le langage C# et développé par Unity Technologies. Il est l'un des plus répandus dans l'industrie du jeu vidéo, aussi bien pour les grands studios que pour les indépendants du fait de sa rapidité aux prototypages et qu'il permet de sortir les jeux sur tous les supports.

Unity propose également une version liée au cloud (gratuite pour les étudiants en faisant la demande) qui permet de se passer de git pour le développement à plusieurs et gère même les conflits lorsqu'un fichier est modifié par plusieurs personnes à la fois

2. Platformer 2D

Les platformers 2D font parti de la grande famille des jeux dits « de plates-formes » dans laquelle on retrouve les premiers jeux « Mario ». Dans les jeux de plates-formes, le joueur contrôle un avatar qui doit sauter sur des plates-formes suspendues dans les airs et éviter des obstacles et ennemis. Les environnements requièrent de devoir sauter ou grimper pour pouvoir être traversés. D'autres manœuvres acrobatiques peuvent venir modifier le gameplay comme l'utilisation d'un grappin ou de trampolines par exemple.

2. Description du jeu

Notre projet est donc un platformer en 2D développé sur Unity version 2019.3.15f1 connectée au « Unity Hub » pour gérer le développement à deux. Nous y avons également connecté visual studio 2019 community pour écrire nos scripts C#.

Voici les principales fonctionnalités du jeu :

- 💻 Un menu jouable contenant un bouton Play, Continue, Settings, Shop et Quit.
- 💻 Un jeu de plates-formes jouable au clavier ainsi qu'avec une manette Xbox connectée au PC.
- 💻 Un système de checkpoint dans les niveaux.
- 💻 Un système de sauvegarde à la fin de chaque niveau.
- 💻 Un système de gestion du volume accessible en jeu.
- 💻 Différentes mécaniques de jeu détaillées dans le chapitre architecture du jeu.
- 💻 Un menu pause standard accessible via la touche escape.
- 💻 Une boutique accessible depuis le menu, donnant accès à différents bonus.

Le but du jeu est donc de se déplacer dans le niveau afin de trouver la porte menant au suivant, en tuant les différents ennemis rencontrés, collectant un maximum de pièces, achetant des bonus permettant de modifier le comportement du joueur et bien évidemment éviter les chutes et les dégâts infligés par les ennemis présents dans les différents niveaux.

3. Architecture du jeu

1. Les mécaniques

Voici donc les différentes capacités de notre héro :

- 💻 Se déplacer sur les plates-formes à l'horizontale.
- 💻 Sauter.
- 💻 Monter aux échelles.
- 💻 Collecter des pièces puis les dépenser au shop.
- 💻 Frapper avec une épée.
- 💻 Lancer des shurikens
- 💻 Bloquer les attaques ennemies.
- 💻 Utiliser un grappin.
- 💻 Activer des leviers et ouvrir des coffres.
- 💻 Traverser des portails.

2. Les objets (gameObjects)

Notre jeu comporte différents objets ou « gameObject » permettant le bon fonctionnement du jeu. Tout objet présent dans le jeu porte sur lui un sous-objet « Transform » qui indique sa position dans la scène (ce qui n'est pas toujours utilisé selon le but de l'objet). Comme les dossiers windows, les gameObjects sont organisés en hiérarchie. Un gameObject peut donc être directement un objet visible dans le jeu ou alors servie de dossier parents pour regrouper différents objets de même type.

Ci-dessous quelques objets principaux de notre jeu que nous gardons entre les niveaux.

Le Player

L'objet « player » est le plus important du jeu car il régit tout ce qui touche au joueur. Cela comprend ses animations, son comportement (déplacements, réaction à l'environnement et combats) son inventaire, son état (utilisé pour la sauvegarde en fin de niveau ainsi que le chargement au lancement du jeu), ou encore sa barre de vie.

L'objet est donc complexe et porte plusieurs composants :

- 💻 Un animator gérant ses animations (idle, mouvements, combat).
- 💻 Un rigidbody 2D, simulant la gravité.
- 💻 Un CapsuleCollider gérant les collisions avec le décor et les ennemis
- 💻 Un Line Renderer & un Distance Joint 2D qui s'activent à l'utilisation du grappin pour gérer la corde.
- 💻 Un script Player Movement gérant les déplacements du joueur.
- 💻 Un script Inventory gérant les pièces du joueur.
- 💻 Un script Player Health gérant la santé et le bouclier du joueur.
- 💻 Un script Player Combat gérant les attaques du player.
- 💻 Un script Player State regroupant certaines données du player pour la sauvegarde.

L'objet player est aussi parent de deux sous-objets : Player Center et VFX. Le premier est utilisé pour différents tests utiles dans les déplacements du joueur, le second pour des effets visuels inhérents au joueur.

La Caméra

La caméra est aussi un objet important puisqu'elle gère l'endroit depuis lequel le jeu est vu et d'où le son est entendu. Sans caméra, le jeu n'est qu'un écran noir sans bruit.

Notre caméra porte donc 4 composants :

- Une caméra gérant la capture de l'image.
- Une 2eme caméra pour le fond qui donne un effet parallaxe
- Un audio listener gérant la capture des sons émis par le jeu.
- Un script Camera Follow gérant les déplacements de la caméra en suivant ceux du joueur.

Grâce à cela notre jeu peu être vu et entendu, et la caméra suit le joueur dynamiquement.

Les Canvas

Les canvas sont des objets utilisés pour gérer l'interface utilisateur. Ils sont transparents et agissent comme un filtre se positionnant juste devant le capteur de la caméra. Ils permettent ensuite de disposer des éléments graphiques à afficher qui seront à des endroits fixes de l'écran. Dans notre jeu un canvas suit en permanence la caméra et nous permet d'afficher au joueur sa barre de vie et de bouclier, son nombre de pièces ainsi que de faire un effet de fondu noir au moment de la mort et du chargement des niveaux. En ce qui concerne la barre de vie et de bouclier, ces deux éléments portent un script permettant de mettre les données à jours via le script du joueur « Player Health ».

Le menu « Pause »

Ce menu est simplement un canvas au premier plan qui est désactivé. Il s'active à la pression de la touche escape, ce qui stop l'écoulement du temps dans le jeu et permet de cliquer sur différents boutons (resume, menu, settings ou save and quit).

L'audio Manager

Cet objet est le centre de gestion des sons de notre jeu. Il ne comporte qu'un seul script qui contient une liste de sons (nous avons créé une classe Sound afin de pouvoir utiliser les sons dans ce script) et une méthode Play. Grâce à cet audio manager, nous pouvons jouer n'importe quel son depuis n'importe quel script du jeu, en une ligne simple de code :

```
FindObjectOfType<AudioManager>().Play("nameOfSound");
```

La liste doit être remplie manuellement en important les sons dans le projet. Au lancement du jeu, la liste est chargée et les sons sont disponibles pour être joués aux bons moments.

L'event System

L'event system est un objet généré par unity qui sert à gérer les événements ; par exemple les clics/glissements de souris sur un slider

Le Game Manager

Cet objet comporte un simple script empêchant la destruction d'objets lorsque nous passons d'une scène à une autre (les scènes servent de niveaux). Le script comporte une liste de gameObject (ceux listés précédemment ici) et au chargement d'une scène, ils ne sont pas détruits. C'est grâce à cela que nous pouvons sauvegarder la progression du joueur, son inventaire ou encore les objets achetés au shop.

Les Tilemaps

Une tilemap est simplement une grille sur laquelle nous pouvons disposer tous les éléments de notre niveau. Du décor à l'arrière plan aux décors disposés devant le joueur en passant par les plates-formes et les échelles, tous les objets immobiles de décor (interactifs ou non). Nous avons donc importé dans notre projet différentes images comportant ces bouts de décor afin de pouvoir créer des niveaux dans différentes ambiances. Afin de pouvoir gérer le rôle de chacune de ces tilemaps, nous changeons leur layer et utiliseront cette spécificité dans nos script, pour effectuer un wall jump, gérer les double / triple saut ou encore grimper aux échelles.

3. Les préfabriqués (prefabs)

Les préfabriqués sont des gameObjects que nous avons créés puis enregistrés (avec tous leurs composants), ce qui permet de les réutiliser facilement sans devoir recommencer de 0 à chaque fois ou devoir utiliser le copier-coller.

Ce procédé nous est très utile pour pouvoir placer des pièces à collecter dans les niveaux, des ennemis ou des éléments de décor tel que des jumpers, plates-formes mouvantes ou leviers et portes

4. Les scènes

Les scènes sont des environnements contenants des éléments du jeu. Pour notre projet nous avons choisi d'utiliser une scène par niveau, plutôt que de n'en avoir qu'une seule géante avec chaque niveau à différents emplacement dans celle-ci. Cela facilite le développement à plusieurs et limite les conflits lors de l'upload des modifications apportées au projet (chacun peut créer sa scène pour y effectuer ses tests sans influer sur le projet final).

Afin de pouvoir utiliser le mangeur de scène et de gérer les changements de scènes dans les scripts (charger une autre scène ou simplement recharger la scène en cours en cas de mort), il faut ajouter un using au début du script :

```
using UnityEngine.SceneManagement;
```

La scène du Menu comporte plusieurs Tilemaps (main menu, settings et shop).
Si le joueur veut entrer dans le shop, la tilemap du menu disparait pour laisser place à celle du shop avec les items à acheter.

Ces changements sont gérés via les scripts "Change Menu" attachés aux boutons de la tilemap active.

Pour les changements de niveaux, c'est une nouvelle scène qui sera chargée.

5. Les scripts

Dans Unity, les scripts s'écrivent en C# (il existe encore la possibilité de les écrire en Javascript mais Unity le déconseille).

Tous les scripts suivent une même architecture et voici donc les parties communes à chacun :

- Les « using » sont les éléments utilisés par le script (collections de classes, Namespaces ou autres types de données).
- Les variables utilisées par le script. Elles peuvent être publiques (accessibles et visible depuis l'environnement de développement ou les autres scripts) ou privées (seul le script peut les utiliser et les voire).
- Les méthodes / fonctions.

4. Les scripts importants

1. PlayerMovement

Il s'agit du script centrale de notre jeu. Il contient énormément de variables, et de méthodes faisant appel à d'autres scripts du jeu.

Les variables horizontalSpeed, verticalSpeed, jumpForce/jumperForce, horizontalMovement et verticalMovement sont utilisées afin de faire bouger le personnage sur axe horizontal ou vertical (course et sauts). La variable rb transmet les vitesses calculées grâce aux autres variables, au personnage en jeu.

airJumpCount, airJumpMax et canWallJump servent à gérer le nombre de sauts disponibles.

Si le joueur ne dispose que d'un saut, la valeur de airJumpMax sera à 0, sinon elle sera égale au nombre de sauts total -1 (elle correspond donc au nombre de saut maximum disponibles en l'air). En combinaison avec airJumpCount (qui compte le nombre de saut au l'air déjà effectué), nous pouvons facilement gérer si le joueur, en l'air, a le droit de sauter à nouveau ou non.

canWallJump, change en fonction du layer avec lequel notre joueur est en contact; si il touche un mûr alors la valeur est vraie, sinon pas de wallJump.

Les variables de type LayerMask servent lors des différents tests afin de savoir si notre joueur est en contact avec un environnement spécifique (le sol pour les sauts, une échelle pour pouvoir y grimper, un mûr pour les walljumps, un jumper pour les sauts plus haut ou encore si un point d'accroche pour le grappin est à portée).

Les variables animator, spriteRenderer, rope et line quant à elles, sont utilisée afin de gérer les changements graphiques liés aux déplacements.

La variable animator permet de passer d'une animation d'attente, "idle" (personnage immobile), à une animation de course ou de saut par exemple (ces changements sont gérés dans le composant animator du gameObject "player").

spriteRenderer sert uniquement à effectuer une symétrie d'axe y, lorsque le sens de déplacement du personnage change.

Quant aux deux dernières :

rope est le lien physique entre 2 points (player et hookspot) et line est sa représentation graphique lorsque le personnage utilise son grappin.

La variable de type vector2 "hookspot", nous permet de stocker la position du point auquel va s'attacher le grappin du joueur, afin de pouvoir tracer une ligne entre ce dernier et le centre du gameObject "player".

Les variables groundCheck, playerCenter, groundCheckRadius et centerCheckRadius servent pour le premier, de points de repères afin de savoir lorsque le personnage touche le sol. Et pour le second, d'avoir un point de départ pour notre grappin.

groundCheckRadius et centerCheckRadius sont simplement les rayons des cercles dans lesquels on effectue ces tests (de simples points étant trop petits, nous les avons agrandis en cercles en augmentant leur radius).

HookShotRangeRadius est le rayon du cercle dans lequel on peut utiliser le grappin (sa portée).

stepsCount est une variable qui compte le nombre de pas (ou plutôt le nombre de frame lorsque le personnage se déplace). Elle nous est utile pour que le son de bruit de pas ne se joue pas à chaque frame mais toutes les 18 frames.

Enfin le booléen canMove sert à empêcher tout mouvement du personnage si, par exemple, une animation ne nécessitant pas de mettre le jeu en pause doit se jouer. Elle évitera au joueur de tomber ou mourir sans même savoir que ses actions n'étaient pas stoppées durant la cinématique.

- Awake : est appellée avant le lancement du jeu.
- onEnable / onDisable : sont appelée à l'activation / désactivation du script.
- setMaxAirJump: permet de fixer le nombre max de saut en l'air.
- Update / fixedUpdate : appelée chaque frame dans le jeu. utilisée pour vérifier les inputs du joueur et, selon les conditions respectées ou non, effectué les actions de déplacement correspondantes.
- Jump : effectue différentes vérifications afin de savoir si le joueur peut ou non sauter. S'il peut sauter, ajoute de la vélocité verticale au rigidBody du joueur et joue un son de saut.
- HookShot : appelle getClosestHookSpot() qui renvoie le point pour grappin le plus proche et active le grappin (si il y en a)
- getClosestHookSpot: détecte tout les points d'accroche de grappin dans une certaine portée, puis renvoie la position du plus proche ou le vecteur nul si aucune point n'est présent.
- HandleAnimation : Modifie les variables d'animation en fonction de l'état / des contacts du personnage.
- HandleHorizontalMovements : Appelée à chaque frame du jeu via fixedUpdate. HorizontalMovement =0, mais lorsque l'on veut déplacer le personnage, la valeur augmente ou diminue progressivement (-0.3 à 0.3) grâce à Time.deltaTime. Selon si le joueur se balance à un grappin ou non la physique change, nous gérons ceci grâce à un test logique. Afin d'éviter de jouer un son de pas lorsque le joueur est en l'air, nous vérifions son contact avec le sol.
- PlayWalkSound : Compte le nombre de frame ou le joueur se déplace afin de synchroniser le bruit de pas avec l'animation. Le son se joue donc toutes les 18 frames ou le joueur se déplace au sol.

- HandleLadder : Appelée à chaque frame du jeu pour vérifier si le joueur est en contact avec un élément comportant le layer “Ladder”. Afin de simuler un effet d’accrochage à l’échelle, la gravité est changée à 0 en cas de contact avec. Puis, si le joueur applique une force verticale au personnage (donc qu’il veut monter ou descendre sur l’échelle), nous modifions la vélocité du personnage de la même manière que pour les déplacements horizontaux.
- isOnJumper, isGrounded, onLadder, isOnWall sont des méthodes retournant un booléen. Nous les utilisons lorsque nous avons besoin de savoir si le personnage est en contact avec un élément de décor particulier.
- OnCollisionExit2D : Lorsque le joueur perd le contact avec le layer numéro 11, correspondant au mûr, il a à nouveau la possibilité d’effectuer un walljump. (pour ne pas pouvoir wall jump 2 fois de suite sur le même mur)
- OnDrawGizmos: sert à rendre visible les différents portés (HookShotRangeRadius, groundCheckRadius, centerCheckRadius) dans la scène.

2. PlayerState

Ce script est parmi les plus importants du jeu puisqu’il permet la sauvegarde de la progression du joueur aux travers de différentes variables :

« currentLevel »; « health »; « shield »; « maxJump »; « maxLevel » et « coins ».

Ce script ne fonctionne pas seul, il nécessite les scripts “SaveSystem” et “PlayerData” pour fonctionner.

Player state est attaché au gameObject Player et se charge de garder à jour les informations sur la progression et l’état du joueur. Il est aussi utile lors du chargement des données précédemment sauvegardées. Au lancement du jeu, un nouvel objet Player est créé, nous vérifions donc si une sauvegarde existe et, si tel est le cas, nous remplaçons les valeurs du nouvel objet avec celles sauvegardées.

PlayerData est une classe sérialisable comportant un constructeur prenant en argument un objet PlayerState (facile☺).

SaveSystem, comporte deux méthodes simples : SavePlayer et LoadPlayer.

SavePlayer prendra en argument un objet PlayerState et l'utilisera afin de créer un objet PlayerData (qui est sérialisable), puis le sauvera dans un fichier binaire d'extension ".bin" à l'emplacement définit dans la méthode.

LoadPlayer renvoie un objet playerData issu de la désérialisation d'un fichier binaire d'extension ".bin". Dans un premier temps, nous vérifions l'existence d'un fichier compatible à l'emplacement choisi, puis effectuons la désérialisation si possible.

3. PlayerHealth & HealthBar

Ce script fonctionne gère la vie du personnage. Il fonctionne en collaboration avec HealthBar et ShieldBar, qui eux gèrent l'affichage des informations de PlayerHealth à l'écran. Leur fonctionnement étant similaire nous ne commenterons que HealthBar.

Dans un premier temps nous initialisons nos variables puis, grâce à la méthode Awake(), nous recherchons dans la scène les scripts permettant l'affichage de nos informations et les affichons. La méthode Awake s'exécutant avant tout le reste, si nous chargeons une sauvegarde, les valeurs de la sauvegarde écraseront celles attribuées ici (les méthodes setHealth et setShield servent à cela).

- TakeDamage (int damage) :
  Ici nous retirons des points de vie ou de bouclier au personnage, lorsque ce dernier subit de dégâts (le montant des dégâts est placé en argument). Si le joueur n'est pas invincible (car il vient de subir des dommages par exemple), nous effectuons différents test afin de savoir si nous devons enlever des points de vie, bouclier, les deux ou alors faire mourir le personnage, et nous effectuons la mise à jour des informations à l'écran.

- Heal(int heal) (idem Shield(int shield)) :
  Sert à rendre des points de vie au personnage, lorsqu'il ramasse un item lui rendant des PV ou qu'il achète ce même item dans le shop, et mettre à jour les informations à l'écran.. Le montant de PV à rendre est en argument.

- ResetLevel() :
  Permet de recommencer un niveau avec toute sa vie (en cas de mort par exemple. Ici, le montant du bouclier n'est pas modifié car il s'agit d'un bonus.

- Coroutine Respawn() :
  Commence par jouer l'animation de mort du joueur, puis effectue un fondu (pour masquer le respawn du joueur au début du niveau), enfin la vie du joueur est remise au max et l'on replace le joueur au point de spawn.

- InvicibilityFlash() & HandleIncincibilityDelay()
  Ces méthodes sont toujours appelées en même temps. La première fait clignoter le personnage pour donner un effet de prise de dégât et donc d'invulnérabilité tant que celui ci est invincible. La deuxième quant à elle, gère le temps d'invincibilité.

Pour ce qui est du script HealthBar, il utilise 3 composants :

- Un slider, qui sert de barre de vie.

- Un gradient, qui sert à changer la couleur de la barre du slider en fonction de sa valeur (vert, jaune et rouge).

- Une image, qui est un petit cœur pour indiquer qu'il s'agit de la santé de notre personnage.

Deux méthodes, une (SetMaxHealth) pour définir la vie maximale du joueur et la lui attribuer (au lancement du jeu), et l'autre (SetCurrentHealth) pour appliquer des changements à la vie actuelle du joueur et donc afficher la bonne valeur au joueur.

4. PlayerCombat

Dans ce script sont gérées toutes les interactions de combat (coup d'épée, lancé de shuriken ninja et super parade).

Il utilise 8 variables :

- playerRb pour savoir dans quelle direction regarde le player. (facing right)
- 3 paires de variables très similaires (« timeBetween... » & « next…time ») servent à attendre un certain temps après avoir fait une action (block, sword & shuriken)
- enemyLayer sert à reconnaitre les ennemis sur les attaques.

Ses méthodes sont :

- takeDamageFromEnemy() :
  Méthode public appelée par les ennemis, si le joueur n'est pas en train de bloquer (dans son animation) il prend des dégâts.
- Update() :
  On actualise facing right.
- Block() :
  Lance l'animation de blocage & son effet.
- AttackSword() :
  Lance l'animation d'attaque.
- didAttackHitEvent() :
  Appelée par un event sur une frame de l'animation d'attaque, pour tous les ennemis "touchés", leur inflige des dégâts.
- ThrowShurkien() :
  Génère un shuriken, lui donne une vitesse selon la direction du joueur.

5. Améliorations possibles

Nous pensons que certains sons pourraient être mieux choisis, mais pour l'instant nous voulions simplement nous assurer que notre système serait fonctionnel, nous aurons tout le loisir de les modifier par la suite.

Nous nous sommes beaucoup concentrés sur l'ajout de mécaniques dans le jeu et dans la création d'éléments préfabriqués, afin de rendre la création de niveau largement simplifiée par la suite, nous n'avons donc pas beaucoup de niveaux pour le moment.

6. Conclusion

Pour réaliser ce projet nous avons été aussi vite que possible mais aussi lentement que nécessaire, ce qui se ressent dans le résultat.

Grâce à cela nous avons pu apprendre à utiliser Unity et son environnement, qui semblent être très puissants, mais aussi parfois assez difficile à appréhender (pour ne pas dire super chiant). Malgré les nombreuses fonctionnalités présentent dans notre jeu, nous savons que nous n'avons touché qu'à une petite partie des possibilités offertes par Unity. Au final nous avons pris beaucoup de plaisir lors de ce projet et sommes fier du résultat obtenu.

7. Bibliographie

https://fr.wikipedia.org/wiki/Jeu_de_plates-formes

https://fr.wikipedia.org/wiki/Unity_(moteur_de_jeu)