Guide du module : Twin-stick Shooter

Objectif

L'objectif de ce module est de recréer la phase de piratage du jeu **Nier:Automata** (2017) développé par **PlatinumGames** et édité par **Square Enix**. Voici une vidéo de référence pour l'exercice : **https://youtu.be/_nkmv1X_VY0**.

Les éléments indispensable à reproduire sont :

- **Les contrôles du joueur**
- **Les ennemis**
- **Les conditions de victoire/défaite**

Progression

1. L'environnement
2. Le personnage joueur
3. Les ennemis
4. Le tir et la destruction des ennemis
5. Conditions de victoire / défaite

L'environnement

Modéliser l'arène

- Pour le sol, créer un **GameObject** de type plan, nommé "Ground". Le redimensionner pour qu'il soit assez grand.
    - Créer un nouvel asset de type **Material** nommé "Ground" et l'assigner au composant **MeshRenderer** du **GameObject**.
- Pour les 4 murs, créer un **GameObject** de type cube. Le nommer "Wall".
    - Créer un nouvel asset de type Material nommé "Wall" et l'assigner au composant **MeshRenderer** du **GameObject**.
    - Dupliquer 3 fois pour obtenir 4 **GameObjects** au total. Les redimensionner et les positionner pour entourer le sol.

La caméra

- Positionner et orienter la caméra pour avoir la même fenêtre que dans la vidéo de référence.
    - Orienter la caméra de la fenêtre **Scene** comme on le souhaite.
    - Avec le **GameObject** **"Main Camera"** sélectionné, utiliser le raccourci CTRL-SHIFT-F ou (CMD-SHIFT-F sur mac) pour aligner le **GameObject** avec la caméra de la fenêtre **Scene**.
    - Enfin, modifier à la main les valeurs du **Transform** de la **Main Camera** pour arrondir les valeurs et centrer proprement.

Le personnage joueur, modélisation

- Créer un **GameObject** de type cube pour le personnage. Le nommer “Player”. Le placer juste au-dessus du sol.
    - Laisser CTRL-SHIFT appuyées pendant le déplacement du **GameObject** pour le faire se coller automatiquement à la surface du sol.
        - Attention, il faut être en mode “Center”, et non “Pivot”.
    - Créer un **Material** “Player” et l’assigner au composant **MeshRenderer**.

- Pour indiquer la direction dans laquelle regarde le personnage (sa direction **forward**, l’axe bleu), on va lui ajouter un nez.
    - Créer un **GameObject** de type cylindre pour son nez, l’orienter et le positionner au niveau de la direction **forward** du cube.
    - Assigner le **Material** “Player” au **MeshRenderer** du nez.

Déplacer le personnage

Le jeu se joue au **gamepad**. Le joystick de droite sert à l’orienter, celui de gauche à se déplacer dans l’arène. La gâchette de droite sert à tirer.
Au clavier, le joystick de gauche est remplacé par les touches ZQSD et celui de droite par les flèches. Tirer se fait avec la barre espace.

1) Déplacer le personnage avec son transform

Déplacer le personnage grâce à la méthode vue en cours (transform.position ou transform.Translate).

2) Déplacer le personnage avec son Rigidbody

Pour déplacer le personnage :

- Ajouter un composant **Rigidbody** sur le **GameObject** du personnage.
- Ajouter un nouveau script nommé “PlayerMovement”.
    - L’ouvrir en double-cliquant dessus.

Initialisation du script

Les scripts Unity possèdent des méthodes particulières, qu’on appelle “Message Unity”. Ce sont des méthodes qui sont appelées automatiquement par Unity, si elles existent. Elles correspondent au cycle de vie d’un **GameObject**.

1. `private void Awake()` : appelée au chargement du script, au tout début de la création de l’objet,
2. `private void Start()` : appelée juste avant la première **frame** après le chargement du **GameObject**,
3. `private void Update()` : appelée à chaque frame,
4. `private void LateUpdate()` : appelée à chaque fin de frame,
5. `private void FixedUpdate()` : appelée à chaque pas de temps physique.

- Dans la méthode **Awake** du script **PlayerMovement**, récupérer une référence vers le composant **Rigidbody** dans une variable globale nommée "_rigidbody" grâce à la fonction **GetComponent** (utiliser la version générique).

Récupérer les inputs

- Dans la méthode **Update** du script, utiliser la classe **Input** pour récupérer les informations du joystick gauche ou des touches ZQSD du clavier. On utilisera la méthode **Input.GetAxisRaw**. On les stockera dans deux variables locales, "**horizontal**" et "**vertical**".
- Créer une variable globale privée de type **Vector3** et la nommer "**_movementInput**". Assigner à cette variable une nouvelle valeur à partir des deux variables locales créées précédemment. On mettra "**horizontal**" dans le champ "**x**" et "**vertical**" dans le champ "**z**". 0 pour le champ "**y**".
- Appliquer la fonction **Normalize** à ce vecteur pour obtenir une **direction**, c'est-à-dire un vecteur de longueur 1.
- Ce vecteur représente la direction vers laquelle le joueur veut déplacer le personnage.

Calculer le vecteur vitesse

Un vecteur vitesse est calculé à partir d'un produit d'un vecteur direction (un vecteur de longueur 1) et d'une vitesse (un **float**).

- Créer une variable globale privée et sérialisée de type **float** que l'on nommera "**_speed**".
    - Pour sérialiser une variable globale privée, utiliser l'attribut **SerializeField**

Dans la méthode **FixedUpdate**, calculer le vecteur de vitesse du personnage :

- Créer une variable locale de type **Vector3** nommée "**velocity**".
- Lui assigner comme valeur le produit entre le vecteur _movementInput et la variable "_speed".

Application du vecteur vitesse

- Enfin, toujours dans la méthode **FixedUpdate**, accéder au composant **Rigidbody** par sa variable "_rigidbody" et assigner à sa propriété **velocity** le nouveau vecteur vitesse "velocity".

Tests et corrections

Tester et régler les éventuels problèmes :

- Le personnage ne bouge pas ?
    - Vérifier dans l'inspecteur que la valeur de Speed n'est pas 0.
    - Ajouter un asset de type **PhysicsMaterial** aux composants **Collider** du Player et du sol avec une friction à 0.
    - Vérifier le code.
- Le cube se met à tourner sur lui-même de manière incontrôlable ?
    - Régler les contraintes de rotation dans le composant **Rigidbody** (**Freeze Rotation**).
- Le cube n'avance pas et ne va pas vers la gauche

    - Vous utilisez le clavier ? Unity est configuré par défaut en qwerty. Il faut modifier les touches dans l'**Input Manager**.

Orienter le personnage

Inputs

L'orientation du personnage se fait soit avec l'axe horizontal du joystick droit du gamepad, soit avec les flèches gauche-droite du clavier. Il faut tout d'abord configurer ces inputs dans les propriétés du projet.

- Il faut créer de nouvelles entrées dans l'**Input Manager** pour le joystick droit et les flèches du clavier. Le plus simple est de dupliquer les entrées existantes.
    - Pour savoir à quel axe correspond le joystick droit, on peut se référer à cette page :
        - **XBox 360**
        - **XBox One**
        - **PlayStation 4 DualShock**
        - Manette freebox
            - Attention, il faut avoir l'option "Analogue" activée
            - Button 1: joystick button 0
            - Button 2 : joystick button 1
            - Button 3 : joystick button 2
            - Button 4 : joystick button 3
            - LB : joystick button 4
            - RB : joystick button 5
            - Left stick X axis : X axis
            - Left stick Y axis : Y axis
            - Right stick X axis : 3rd axis
            - Right stick Y axis : 5th axis
        - Si vous ne trouvez pas le mapping pour votre manette, vous pouvez utiliser cet outil **Unity Input Displayer** (utiliser Chrome).
    - Pour les flèches du clavier, il faut en plus supprimer les touches dans les entrées "Horizontal" et "Vertical", dans "Positive Button" et "Negative Button".
- Attention aux propriétés **Gravity**, **Dead**, **Sensitivity** et **Snap** qui sont différentes selon le type d'input (une touche du clavier, ou un joystick du gamepad). Il faut suivre les valeurs des entrées déjà existantes.

Récupérer les inputs dans le script

- Dans la méthode **Update** du script PlayerMovement, à la suite du code qui gère le mouvement, récupérer les informations du joystick droit ou des flèches du clavier et les stocker dans deux variables locales, "**orientationHorizontal**" et "**orientationVertical**".
- Créer une variable globale privée de type **Vector3** et la nommer "**_orientationInput**". Assigner à cette variable une nouvelle valeur à partir des deux variables locales créées précédemment. On mettra "**orientationHorizontal**" dans le champ "x" et "**orientationVertical**" dans le champ "z". 0 pour le champ "y".

Ce vecteur représente la direction dans laquelle le joueur veut faire viser le personnage.

Calculer la rotation

Dans Unity, une **rotation** peut être calculée de différentes façons. Soit avec des angles en degré que l'on appellera eulériens, soit avec des objets mathématiques nommés **Quaternion**. Ici, on utilisera des **Quaternion** car ils offrent de nombreuses possibilités pour les calculs.

Dans la méthode **FixedUpdate**, calculer le quaternion rotation du personnage :

- Créer une variable locale de type **Quaternion** nommée "**lookRotation**".
- Lui assigner comme valeur la fonction **Quaternion.LookRotation** appliquée au vecteur **_orientationInput** calculé précédemment.

Appliquer la rotation

- Enfin, toujours dans la méthode **FixedUpdate**, accéder au composant **Rigidbody** par sa variable "_rigidbody" et utiliser la méthode **MoveRotation** pour appliquer la nouvelle rotation "lookRotation".

Tests et corrections

- J'ai une erreur "**Input XXX is not setup**"
  - Vérifier que l'on a bien créé l'entrée dans le tableau des inputs dans l'Input Manager.
  - Vérifier que l'orthographe entre la propriété "Name" et le paramètre que l'on donne à **GetAxisRaw** est strictement identique.
- J'ai une erreur "**Rotation quaternions must be unit length**"
  - Cette erreur vient du fait que lorsqu'on n'appuie pas sur le joystick droit, on donne à la fonction **LookRotation** un vecteur de longueur 0, ce qui n'est pas autorisé.
  - Il faut d'abord tester la longueur du vecteur "_directionInput" avant de la donner à la fonction. Utiliser la propriété **sqrMagnitude**.
    - Si la valeur est supérieure à un certain seuil, alors on calcule la rotation.
    - Sinon, on ne fait rien.

Les ennemis

Dans la séquence de hacking de Nier:Automata il y'a plusieurs types d'ennemis :

- Celui qui fonce vers le joueur en tirant,
- Celui qui tourne sur lui même en tirant,
- Celui qui fonce vers le joueur (sans tirer).

Dans un premier temps, nous allons nous concentrer sur le dernier type, l'ennemi qui fonce vers le joueur.

Création du prefab

On va créer un modèle pour l'ennemi (asset de type **Prefab**) à partir duquel on créera tous les ennemis de la scène (instances GameObject).

- Créer un nouveau **GameObject** de type cube dans la scène. Le nommer "Enemy". Ajouter un composant **Rigidbody** dessus. Cocher les contraintes de rotation.
- Créer un dossier "**Prefabs**" dans la fenêtre **Project**.
- Cliquer-glisser ce **GameObject** de la fenêtre **Hierarchy** vers la fenêtre **Project**, dans le dossier Prefabs. Cela crée l'asset de type **Prefab** qui prend le même nom que le **GameObject**.
- Supprimer le **GameObject** Enemy de la fenêtre **Hierarchy**.
- Double-cliquer sur le **Prefab** Enemy dans la fenêtre **Project** pour passer en "**mode Prefab**".
- Réinitialiser la position du **Transform** en (0, 0, 0).

On peut ensuite customiser cet ennemi :

- Ajout d'un "nez". Soit un cylindre comme pour le Player, soit un autre cube avec une rotation par exemple.
- Ajout d'un **Material** pour changer sa couleur (par exemple en rouge).

Une fois le **Prefab** préparé, on peut commencer à placer des instances de ce **Prefab** dans la scène.

- Cliquer-glisser le **Prefab** de la fenêtre **Project** à la fenêtre **Hierarchy** pour créer une instance. On peut ensuite placer l'instance comme on le souhaite.

ou

- Cliquer-glisser le **Prefab** de la fenêtre **Project** directement dans la fenêtre **Scene** pour créer une instance.

Placez ainsi au moins 5 ennemis dans l'arène.

Faire tourner l'ennemi vers le joueur

On veut que cet ennemi se tourne vers le joueur à une certaine vitesse. Il ne faut pas que cela soit instantané, sinon le joueur ne pourra jamais les éviter.

- Attacher un nouveau script "**EnemyWalker**" au **Prefab** Enemy.

Voici les éléments dont on aura besoin pour ce script :

- La position actuelle du personnage joueur,

- La position actuelle de l'ennemi,
- La rotation actuelle de l'ennemi,
- La vitesse de rotation.

Position actuelle du personnage joueur

La position actuelle du personnage joueur est donné par le composant **Transform** du **GameObject**. Il faut créer une référence vers ce composant dans le script EnemyWalker.

- Créer une variable globale sérialisée de type **Transform** nommée "_playerTransform".
- Dans l'inspecteur, faire glisser le **GameObject** Player dans ce champ.
    - Attention ! On ne peut pas référencer un **GameObject** de la fenêtre **Hierarchy** depuis un **Prefab**. Il faut que cela soit fait obligatoirement depuis une instance du **Prefab**, appartenant elle-aussi à la **Hierarchy**.

Position actuelle de l'ennemi

La position actuelle de l'ennemi est donnée par la propriété **position** du composant Transform du **GameObject**. On peut accéder directement au composant **Transform** par la propriété **transform**.

- Attention ! Accéder au Transform par ce raccourci fait appel à la méthode **GetComponent**. Il vaut mieux alors toujours mettre en cache la valeur dans une variable globale, dans la méthode **Awake**, comme on l'a fait pour le composant **Rigidbody**.

Rotation actuelle de l'ennemi

La rotation actuelle de l'ennemi est donnée par la propriété **rotation** du composant Transform du **GameObject**.

Vitesse de rotation

La vitesse de rotation doit être paramétrée depuis l'inspecteur. Il faut donc créer dans le script une variable globale sérialisée de type float pour la faire apparaître dans l'inspecteur.

Calcul de la rotation

A chaque pas de temps, on va calculer la nouvelle rotation que doit prendre l'ennemi. Il va s'agir d'une fraction de la rotation de l'ennemi vers la direction où se trouve le joueur.

- Créer une nouvelle méthode nommée "**TurnTowardsPlayer**" sans argument (ou paramètre) et qui retourne **void**.
- Calculer la direction vers laquelle se trouve le joueur,
- Utiliser la fonction **Quaternion.LookRotation** avec cette direction,
- Utiliser la fonction **Quaternion.RotateTowards** avec la rotation actuelle de l'ennemi et la rotation précédente.

Application de la rotation

- À la suite, et utiliser la méthode **MoveRotation** pour appliquer la nouvelle rotation.
- Appeler la méthode "TurnTowardsPlayer" dans la méthode **FixedUpdate**.

Faire avancer l'ennemi

Pour faire avancer l'ennemi, c'est beaucoup plus simple : il va tout droit, à une certaine vitesse. C'est sa rotation qui fait qu'il va avancer vers le joueur.

- Reprendre ce qui a été fait pour faire avancer le personnage joueur, en prenant comme direction pour le vecteur vitesse la direction **forward** de l'ennemi.
    - La direction **forward** est donnée par la propriété **forward** du composant **Transform** du **GameObject**.

Faire tirer le personnage

On va faire tirer le personnage en 4 étapes :

1. Créer un **Prefab** de balle.
2. Instancier ce modèle à chaque fois que l'on veut créer une balle.
3. Faire avancer la nouvelle balle tout droit.
4. Régler la cadence de tir.

Le modèle de balle

- Créer un **GameObject** de type sphère, le nommer "**Bullet**". Le redimensionner.
- Ajouter un composant **Rigidbody**
    - Cocher "**Is Kinematic**".
    - Décocher "**Use Gravity**".
    - Cocher toutes les contraintes de rotation.
- Créer un **Prefab** à partir de ce **GameObject**.
    - Cliquer-glisser le **GameObject** Bullet de la fenêtre **Hierarchy** vers la fenêtre **Project**, dans le dossier "Prefabs".
- Supprimer le **GameObject** Bullet de la scène.
- Avec le **Prefab** sélectionné, réinitialiser la position du **Transform** à (0, 0, 0) dans l'inspecteur.

Créer une balle

Tant que la gâchette de droite du gamepad (**right trigger**) reste appuyée, ou la touche espace du clavier, le personnage tire.

Inputs

D'abord, les inputs :

- Créer les entrées correspondantes dans l'**Input Manager**.
    - Attention, la gâchette de droite du gamepad est de Type "Joystick Axis" tandis que la touche espace du clavier est de type "Key or Mouse Button".

Spawn

Ensuite, le point d'apparition (**spawn**) de la balle. On a besoin de savoir où la balle va apparaître. On pourrait utiliser la position du personnage, mais cela la ferait apparaître à l'intérieur du personnage, ce qui créerait des collisions inutiles.

- Créer un **GameObject** vide, enfant du **GameObject** Player, et le nommer "**Cannon**".
- Positionner le **GameObject** au bout du nez du personnage, de façon à ce qu'une balle créée à cet endroit ne soit pas en collision.

Script

Enfin, le script :

- Attacher un nouveau script nommé "**PlayerShoot**" au **GameObject** Player.

On va créer une référence vers le modèle de balle dans le script PlayerShoot.

- Déclarer une variable globale sérialisée de type **GameObject** nommée "**_bulletPrefab**".
- Lui assigner le **Prefab** Bullet en faisant glisser ce dernier depuis la fenêtre **Project** vers la fenêtre **Inspector**, dans la propriété "**Bullet Prefab**" qui apparaît dans le composant "**Player Shoot (Script)**".

On a maintenant besoin d'une référence vers l'endroit où faire apparaître la balle. On rappelle que la position dans le monde est comprise dans le composant **Transform** du **GameObject**.

- Déclarer une variable globale sérialisée de type **Transform**, nommée "**_cannon**".
- Lui assigner le **GameObject** Cannon en le faisant glisser depuis la fenêtre **Hierarchy** vers la fenêtre **Inspector**, dans le champ "**Cannon**" qui apparaît dans le composant "**Player Shoot**".

Enfin, un peu de code.

- Créer une méthode privée nommée "**FireBullet**", sans paramètre et qui retourne **void**. C'est dans cette méthode que nous allons mettre le code de création de la balle.
- Dans cette méthode, utiliser la méthode **Instantiate** pour créer une nouvelle balle à partir du **Prefab** _bulletPrefab et de la position et la rotation du canon (voir la propriété **transform**). Mettre la valeur obtenue dans une variable locale "**newBullet**".

- Dans la méthode **Update**, faire un test en utilisant la fonction **Input.GetAxisRaw** pour la gâchette droite, ou **Input.GetButton** pour la barre espace pour déterminer quand le joueur veut tirer.
    - Appeler la méthode **FireBullet** si le test retourne **true**.

Faire avancer la balle

Les balles doivent avancer droit devant elles à partir du moment où elles sont créées dans la scène. Nous allons créer un script attaché à la balle qui va se charger de cela. La méthode employée sera de calculer à chaque pas de temps la nouvelle position que la balle doit avoir, et de la déplacer à cet endroit.

Sélectionner le **Prefab** Bullet dans la fenêtre **Project**. Dans l'inspecteur, attacher un nouveau script nommé "**Bullet**".

- Ce script a besoin :

    - D'une variable contenant une référence vers le composant **Transform** de la balle,
    - D'une variable contenant une référence vers le composant **Rigidbody** de la balle**,**
    - D'une variable qui représente la vitesse de la balle "_bulletSpeed".
- Dans la méthode **Awake**, récupérer les références vers les différents composants (Transform et Rigidbody).
- Dans la méthode **FixedUpdate**,
    - Calculer le vecteur vitesse de la balle "velocity". On utilisera la propriété **forward** du composant **Transform** ainsi que la variable vitesse de la balle.
    - Calculer la quantité de mouvement "movementStep" effectué à cette frame. Il suffit de multiplier le vecteur vitesse par la propriété **Time.fixedDeltaTime**.
    - Calculer la nouvelle position de la balle "newPos". On partira de la **position actuelle** de la balle à laquelle on ajoutera la quantité de mouvement.
    - Utiliser la méthode **MovePosition** du rigidbody en lui donnant en paramètre la nouvelle position de la balle.
- Créer une nouvelle méthode public nommée "**Shoot**" qui retourne **void** et prend comme paramètre un **float** nommé "**speed**".
    - Cette méthode est **public** car on l'appellera d'un autre script.
    - Dans cette méthode, assigner à la variable "**_bulletSpeed**" le paramètre "**speed**" de la méthode.

Retournons dans le script **PlayerShoot**.

- Créer une variable globale sérialisée de type float nommée "**_bulletSpeed**".
- Après l'instanciation de la balle, récupérer son composant **Bullet** dans une variable locale grâce à la méthode **GetComponent** sur la variable **newBullet**.
- Appeler la méthode Shoot sur le composant en lui passant comme argument la variable **_bulletSpeed**.

Régler la cadence de tir

Le problème restant est que les balles sont créées à chaque frame ! On va créer une cadence de tir.

- Créer une variable globale privée et sérialisée de type **float** que l'on nommera "**_delayBetweenShots**". C'est le délai entre deux tirs.
- Créer une variable privée de type float que l'on nommera "**_nextShotTime**". C'est la date à partir de laquelle on a le droit de tirer (date == le temps passé depuis le lancement du jeu)

- Dans la méthode **Awake**, assigner à **_nextShotTime** la valeur **Time.time**. C'est la date actuelle. Au moment du démarrage du jeu, Time.time vaut 0.

- Dans la méthode **Update**, créer une condition testant si **Time.time** est supérieure ou égale à **_nextShotTime**.
    - Si oui, alors on peut appeler la méthode **FireBullet**.
        - Il faut également assigner à **_nextShotTime** la prochaine date où on peut tirer, c'est-à-dire la date actuelle à laquelle on ajoute le délai entre les tirs.
    - Sinon on ne fait rien (=> il n'y a pas de else).

Destruction d'une balle

Pour l'instant, les balles créées continuent indéfiniment leur chemin. Au bout d'un moment, il y aura tellement de balles que la mémoire de la machine sera saturée ! On va ajouter du code qui permet de détruire la balle au bout d'un certain délai.

- Dans le script **PlayerShoot**, ajouter une variable sérialisée pour paramétrer le délai avant la destruction de la balle.
- Toujours dans ce script, après avoir tiré la balle dans la méthode FireBullet, appeler la méthode **Destroy** avec en paramètre la propriété **gameObject** de la balle et le délai défini dans l'inspecteur.

Collisions

Collisions entre un ennemi et le personnage joueur

Le but de l'ennemi est de foncer sur le personnage du joueur pour lui rentrer dedans. Nous allons donc paramétrer le système de collisions de Unity pour détecter quand un ennemi touche le joueur.

Il y a plusieurs façon de faire. Pour cette collision, nous utiliserons des **Tags**.

- Donner un **Tag** "Player" au **GameObject Player**.
- Donner un **Tag** "Enemy" au **Prefab Enemy**.

Le moteur physique de Unity calcule les collisions grâce aux composants **Collider** sur les **GameObjects**, ainsi qu'aux composants **Rigidbody**. Cette page explique comment Unity calcule les collisions en fonction des composants présents : **Colliders Overview**.

Dans notre cas, le **Player** et l'**Enemy** ont chacun un **Collider** et un **Rigidbody** dynamique.

Il faut décider maintenant si on veut que les GameObjects puissent rentrer l'un dans l'autre (**Trigger**), ou bien si le moteur physique va calculer une force de répulsion lors de la collision (**Collision**). Nous sommes dans le 2e cas, il n'y a rien d'autre à faire.

Maintenant, pour détecter le moment où la collision a lieu, nous devons écrire du code.

- Créer un nouveau script nommé "**EnemyCollisions**" et l'attacher au **Prefab** Enemy.

Le moteur physique utilise différentes méthodes du script pour envoyer des messages de collision. Comme pour les méthodes de cycle de vie **Awake**, **Start**, **Update**, etc, elles ne sont appelée que si elles existent.

- Créer la méthode **OnCollisionEnter**. Cette méthode est appelée à la première frame où une collision est détectée.
- Dans le corps de cette méthode, il faut tester si l'objet avec lequel l'ennemi est entré en collision (le paramètre collision) est bien le joueur. Pour ce faire, on va tester la valeur de son **Tag** grâce à la fonction **CompareTag**.
    - Si c'est bien le joueur, pour le moment, nous allons juste afficher un message dans la console, avec la fonction **Debug.Log**.

Collisions entre un ennemi et la balle

Le **Prefab** Bullet possède un composant **Collider** ainsi qu'un **Rigidbody Kinematic**. Une **Collision** est possible avec un ennemi selon la **matrice des collisions**.

- Ajouter un **Tag** "Bullet" au **Prefab** Bullet.
- Reprendre le script **EnemyCollisions**.

- Dans la méthode **OnCollisionEnter**, ajouter à la suite du code de la collision avec le joueur, un test pour comparer le **Tag** de l'objet avec l'ennemi est entré en collision avec le **Tag** Bullet.
    - Si le test est positif, alors détruire le **GameObject**. Utiliser la méthode **Destroy**.

Collisions de la balle

On aimerait que la balle disparaisse dans le cas où elle rentre en collision avec un ennemi, mais également lorsqu'elle rentre en collision avec un des murs de l'arène.

- Créer un nouveau script "**BulletCollisions**" et l'attacher au **Prefab Bullet**.
- Créer la méthode **OnCollisionEnter**.
    - Dans son corps, ne pas faire de test, et détruire directement le **GameObject**.

Etudions le cas des murs. Les murs n'ont pas de composant **Rigidbody**, et la balle n'a qu'un **Rigidbody Kinematic**. Une collision n'est donc pas possible pour le moteur physique. Nous allons utiliser un **Trigger**.

- Ouvrir le **Prefab** Bullet
- Cocher "**Is Trigger**" dans le composant **Sphere Collider**.

- Dans le script **BulletCollision**, créer la méthode **OnTriggerEnter**.
- Dans son corps, simplement détruire le **GameObject**.

Le **Collider** de la balle étant devenu un **Trigger**, les messages de collision ne sont plus appelés, seulement les messages de trigger.

- Dans le script **BulletCollision**, supprimer la méthode **OnCollisionEnter**.
- Dans le script **EnemyCollision**, déplacer le code concernant la balle vers une nouvelle méthode **OnTriggerEnter**.
    - Attention, le paramètre entre les deux méthodes **OnCollisionEnter** et **OnTriggerEnter** n'est pas identique...

Condition de victoire/défaite

Un des intérêts de ce jeu vient du fait qu'on puisse gagner et perdre. Dans cet exercice, pour gagner, nous devons tuer tous les ennemis présent dans le niveau et pour perdre le joueur doit se faire toucher trois fois.

Les points de vie du joueur

Pour compter les points de vie du personnage joueur, nous allons utiliser un **ScriptableObject**. Un **ScriptableObject** est un script Unity qui n'est pas un composant. Il n'est pas rattaché à un **GameObject**, mais à des fichiers **Asset**. Il est le plus souvent utilisé pour représenter des données accessibles de partout dans le code.

Le script

- Créer un nouveau script nommé "**IntVariable**" dans le dossier **Scripts**.
- Modifier **MonoBehaviour** par **ScriptableObject**.
- Ajouter l'attribut **[CreateAssetMenu]** devant la définition de la classe
- Ajouter une variable globale public de type **int** nommée "**Value**".

Les instances

- Créer un dossier **Data**.
- Créer dans ce dossier un nouvel asset de type **IntVariable** et l'appeler "**PlayerStartHP**".
    - Dans l'inspecteur, mettre la valeur de la propriété **Value** à 3.
- Créer dans ce dossier un nouvel asset de type **IntVariable** et l'appeler "**PlayerCurrentHP**".

La gestion des points de vie

- Attacher au **GameObject Player** un nouveau script appelé **PlayerHealth**.
- Ajouter deux variables globales sérialisées de type **IntVariable** nommée "**_playerStartHP**" et "**_playerCurrentHP**".
    - Dans l'inspecteur, remplir les nouveaux champs en faisant glisser les deux assets créés précédemment.

À chaque démarrage, on veut que les PVs du joueur reviennent à leur valeur de départ.

- Dans la méthode **Awake**, assigner au champ **Value** de la variable **_playerCurrentHP** la valeur du champ Value de la variable **_playerStartHP**.

Perte de points de vie

Le personnage du joueur perd un point de vie à chaque fois qu'un ennemi entre en collision avec lui.

- Dans le script **EnemyCollisions**, ajouter une variable globale sérialisée de type **IntVariable** et l'appeler "**_playerCurrentHP**".
    - Dans l'inspecteur, remplir ce champ en faisant glisser l'asset **PlayerCurrentHP**.
- À la place de l'écriture d'un message dans la console, décrémenter la valeur du champ **Value** de la variable **_playerCurrentHP**.

Le nombre d'ennemis en jeu

Pour compter le nombre d'ennemis en jeu, nous allons utiliser également une instance de **ScriptableObject** du même type que pour les points de vie du joueur.

Instance

- Dans le dossier **Data**, créer un nouvel asset de type **IntVariable**, nommé "**EnemyCount**".

Nombre d'ennemis en jeu

Il faut savoir quelle valeur mettre dans cette variable au début du niveau. Nous allons créer un script pour compter les ennemis.

- Créer un **GameObject** vide "**EnemyManager**".
- Lui attacher un script "**EnemyManager**".
- Dans ce script, créer une variable globale sérialisée de type **IntVariable**, nommée "**_enemyCount**".
    - Dans l'inspecteur, faire glisser l'asset EnemyCount dans ce champ.
- Dans la méthode **Awake**, assigner à une variable locale le nombre d'ennemi présents dans la scène grâce à la fonction **GameObject.FindGameObjectsWithTag** et au champ **Length** du tableau en retour de cette fonction.
- L'assigner au champ **Value** de la variable **_enemyCount**.

Destruction d'un ennemi

À chaque destruction d'un ennemi, il faut décrémenter la valeur de cet asset.

- Dans le script **EnemyCollisions**, créer une variable globale sérialisée de type **IntVariable**, nommée "**_enemyCount**".
    - Dans l'inspecteur, faire glisser l'asset **EnemyCount** dans ce champ.

À la destruction du **GameObject**, Unity appelle la méthode **OnDestroy** si elle existe.

- Créer la méthode **OnDestroy**, et dans son corps, décrémenter la valeur du champ **Value** de la variable **_enemyCount**.

Fin du jeu

Maintenant que nous avons toutes les valeurs dont nous avons besoin, nous pouvons écrire le code qui va déterminer quand on perd et quand on gagne.

- Créer un nouveau **GameObject** vide "**LevelManager**".
- Lui attacher un nouveau script "**LevelManager**".

- Dans ce script, créer deux variables globales sérialisées pour les points de vie actuels du joueur et pour le nombre d'ennemis encore en jeu.
    - Ne pas oublier de faire glisser les deux assets correspondant dans les champs de l'inspecteur.
- Créer deux méthodes privées **Win** et **Lose** qui ne prennent pas de paramètre et qui retournent **void**.

    - Ces deux méthodes vont simplement écrire dans la console que le joueur a gagné ou perdu.

- Dans la méthode **Update**, on va tester les conditions de victoire et de défaite :
    - Si le nombre d'ennemis atteint 0 ou moins, alors on a gagné, appeler la méthode **Win**.
    - Si les points de vie du joueur atteignent 0 ou moins, alors on a perdu, appeler la méthode **Lose**.

Pour aller plus loin...

Vous avez fini en avance ? Voici quelques idées pour aller plus loin :

- Ajouter un écran Titre au démarrage du jeu
- À la place des messages dans la console en cas de victoire et de défaite, ajouter un écran de Game Over avec des crédits et un bouton pour retourner à l'écran Titre
- Créer un ennemi "boss" qui tire en rond et qui ne peut être attaqué que lorsque tous les ennemis basiques sont détruits
- Ajouter des sons (tir, destruction d'ennemi, perte de point de vie…)
- Créer de nouveaux types d'ennemi :
    - Ennemi avec bouclier, qu'on doit détruire en passant dans son dos
    - Ennemi qui tire
- Ajouter des apparitions aléatoires d'ennemis
- Ajouter un temps limité, l'afficher dans une interface graphique