Workshop Unity3D

Réalisation d'un simple FPS

Avant tout:

1. Installez Unity3D si ce n'est pas déjà fait
2. Créez un nouveau projet 3D.
3. Importez les packages fournis (ici) via Assets->import package

OBJECTIFS: Un simple FPS, en deux étapes

1. Editeur (Prefab, Animation, environnement)
2. C# (Changement de Scène, Joueur, Ennemis, Zone de Spawn)

Dans le dossier workshop, il y a deux dossier: Done/ et To_Do/. Le dossier To_Do/ contient des textures, des sons, et des objets 3D utiles à l'avancement du workshop, toutefois si une partie de l'Editeur ne vous intéresse pas, vous pouvez aller chercher la partie en question dans le dossier Done/ (uniquement la partie Éditeur).

Partie 1: Environnement

- Réalisation de l'environnement:
    - Objet -> Cube pour créer un Cube dans la Scène.

Pour bouger un objet dans la scène: utilisez les flèches (axe en haut à gauche).

Utilisez les différents outils pour créer votre environnement. Vous aurez besoin de plusieurs cubes afin de créer une petite arène.

- Ajouter des textures aux murs:
    - Utilisez les "Materials"

Pour ajouter un Material sur un Objet, il faut simplement glisser le Material dessus.

- Ajouter un FirstPersonController:
    - Simplement glisser la Prefab FirstPersonController dans la hiérarchie, puis le positionner au dessus du sol.

La Prefab FirstPersonController est directement intégrée dans Unity, il est tout de même possible de la recréer simplement. Dans ce workshop nous utiliserons la prefab donnée car elle est très complète. Pour les intéressés :ici

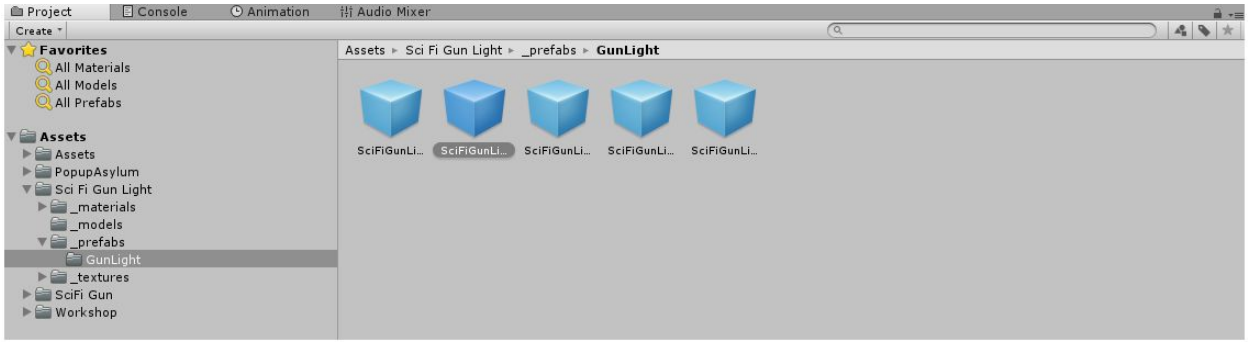
- Ajouter des armes:
    - Une prefab d'arme est déjà réalisée, prête à l'emploie. Dans un premier temps, ajouter cette arme au jeu.

Pour ce faire, glisser l'arme dans la scène, et positionnez la de façon à ce que la Caméra la voit. Vous pouvez utiliser une vue Scène et Game en simultané temps pour voir les modification en temps réel.

Si vous lancez le jeu, vous remarquerez que votre arme est restée sur place, elle ne bouge pas avec vous. La raison est que l'arme n'est associée à aucun parent. Pour ce faire, allez dans l'onglet hiérarchie, et glissez votre arme sur l'objet Camera.

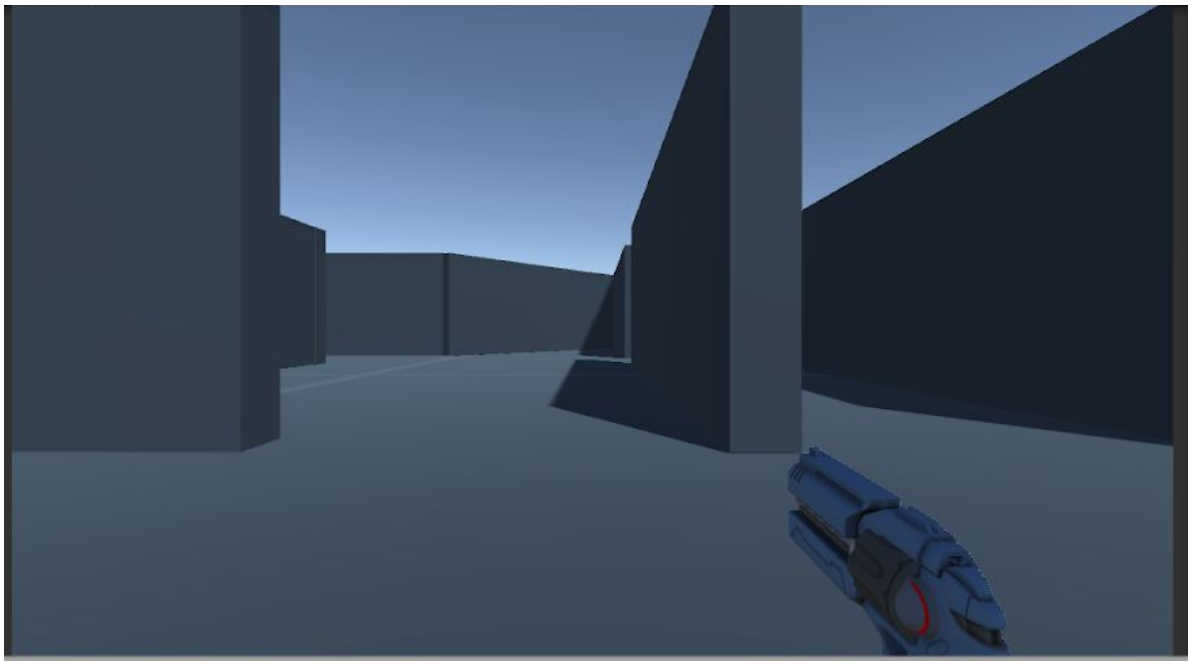
De cette façon, toute les translations et rotations effectuées par la Caméra vont se répliquer sur chacun de ses enfants, à savoir l'arme.

Maintenant que vous avez un petits pistolet, à vous de refaire la même chose avec une arme de plus gros calibre (clic droit -> Create -> Prefab dans l'onglet Project).



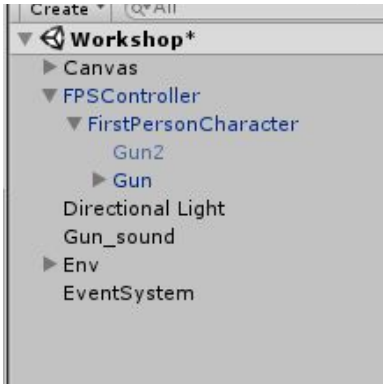
Choisissez une arme, et positionnez la comme le pistolet. Il faudra ensuite faire l'animation (Attention ne pas utiliser un objet Animator mais un Objet Animation).

Regardez la prefab Gun pour vous inspirer. Au final, votre résultat devrait se rapprocher de cela:



*

Votre “Hierarchy” avoir ressembler à cela.



Partie 2: C#

Le c# est un langage orienté Objet, Unity utilise une Scripting API : API

- Premier Script, GameObject, Input

Dans votre objet FPSController, cliquez sur “Add Component” et créez un nouveau script C#.

Les deux fonctions déjà présente dans le script sont Start(), appelée au lancement du jeu, et Update(), appelée à chaque frame du jeu.

Afin de récupérer les actions de l’utilisateur, nous allons utiliser la class Input (class Input). Voici un exemple de Switch:

```
public class Player : MonoBehaviour {
    public GameObject[] weapons;

    private GameObject weapon_active;


    private void Start()
    {
        weapon_active = weapons [0];
    }

    private void Update()
    {
        if (Input.GetKeyDown (KeyCode.A)) {
            Switch_weapon ();
        }

    }

    private void Switch_weapon()
    {
        if (weapon_active.name == "Gun") {
            weapon_active = weapons [1];
            weapons [0].SetActive (false);
        } else if (weapon_active.name == "Gun2") {
            weapon_active = weapons [0];
            weapons [1].SetActive (false);
        }
        weapon_active.SetActive (true);
    }
```

- Le type GameObject est le type de base de toutes les entitées sous Unity3D. La variable weapons est un tableaux de GameObject public, c’est à dire qu’il est possible de donner des valeurs à cette variable directement dans l’éditeur (sur l’objet contenant le script).
- On garde une référence à l’objet actuelle via la variable weapon_active.
- La fonction Switch_weapon() va seulement modifier l’objet de référence à partir des différents objets donnés dans l’éditeurs.

- Animation, type IEnumerator, Coroutine

Vous allez maintenant créer une fonction afin d'animer les tirs des armes. Le "Gun" tirera toutes les 0.75 secondes tandis que le "Gun2" tirera toutes les 0.1 secondes.

Pour ce faire on utilise une fonction de type IEnumerator. Cette fonction est une Coroutine, on peut donc stopper son exécution avec "yield". Ceci vous permet de créer différents types de tir.

→ Comment stopper l'exécution d'une fonction

Pour jouer une animation, utilisez la référence weapon_active afin d'accéder à l'objet Animation:

$$object.GetComponent < Component\ type > ().method;$$

Vous pouvez rajouter un effet sonore à l'aide d'une variable AudioSource publique.

$$AudioSourceObject.Play();$$

Une fois la fonction créée, appelez la dans la fonction Update (avec un Input) à l'aide de StartCoroutine(coroutineFunction). Utilisez un boolean afin de savoir si la coroutine est libre ou non.

Voilà une petite aide, (shoot in game est l'étape suivante) :

```
private void Update()
{
    // if ( ? ) { StartCoroutine(Fire()) }
}

private IEnumerator Fire()
{
    //Set boolean to true (coroutine not free)
    // Play shoot sound
    //Play animation of weapon_active
    //shoot in game
    yield return new WaitForSeconds ((weapon_active.name == "Gun") ? 0.75f : 0.1f);
    //Set boolean to false (coroutine free)
}
```

- Shoot, Rigidbody, Raycast

Pour tirer un projectile, il existe plusieurs solutions; dans Unity il y en a 2 efficaces.

La première utilise des **Rigidbody**. Un Rigidbody est un component que l'on ajoute aux objets et qui a pour effet de contrôler les déplacement de celui ci via une simulation physique en temps réel (gravité, collisions plus ou moins rebondissante, prise en compte de la masse, etc…). Utiliser des rigidbody donne un caractère réel au tir (courbure de la balle, possibilité de tirer plus ou moins loin / fort). Cependant cette technique à un défaut, elle demande beaucoup de ressources, et ne marche pas à tout les coups. En effet le moteur physique doit calculer l'attraction terrestre et les collisions de chaque balle que vous tirez avec l'intégralité des objets de la scènes (y compris les balles entre elles).

Voici tout de même le script utilisant des Rigidbody:

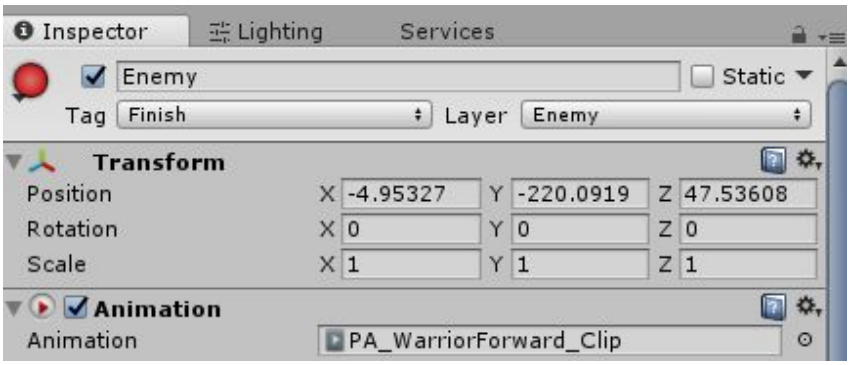
```
public class Player : MonoBehaviour {
    public GameObject[] weapons;

    private GameObject weapon_active;
    public GameObject bullet;
    public Transform bullet_spawn;


    private void Start()
    {
        weapon_active = weapons [0];
    }

    private void Update()
    {
        // if ( ? ) { StartCoroutine(Fire()) }
    }

    private IEnumerator Fire()
    {
        //Set boolean to true (coroutine not free)
        // Play shoot sound
        //Play animation of weapon_active
        //shoot in game
        GameObject bullet_tmp = Instantiate
            (bullet, bullet_spawn.position, bullet_spawn.rotation) as GameObject;

        bullet_tmp.GetComponent<Rigidbody> ().velocity = bullet_tmp.transform.forward * 100;
        yield return new WaitForSeconds ((weapon_active.name == "Gun") ? 0.75f : 0.1f);
        //Set boolean to false (coroutine free)
    }
```

Instanciate() créer un objet dans la scène au position données.

La variable bullet_spawn est un ensemble de coordonnées 3D (position, rotation, scale).

⇒ Pourquoi le Raycast est meilleur ?

Le Raycast utilise également le moteur physique d'Unity3D, cependant aucun objet n'est recréer durant son utilisation. On se contente seulement de tracer une ligne imaginaire (un Vector3). La vérification des collisions est également plus performante car elle permet de choisir un layer d'objet, et ne prend en compte aucune caractéristique physique.



Le layer de la prefab Enemy est Enemy. Ce qui veut dire que le Raycast considérera une collision uniquement avec ce layer (ce layer est généralement le layer 8).

Créez trois nouvelles variables:

```
private Camera child_camera;
private RaycastHit  hit;
private Ray ray;
private int layer = 1 << 8;
```

- child_camera est la caméra de la scène (vous pouvez la passer en public).
- hit permet de stocker l'objet en collision
- ray représente la ligne tracée par le Raycasting.
- 8 représente le layer Enemy, le seul à prendre en compte. Si on veut que layer contienne tous les layer sauf le 8, il faudrait ajouter "layer = ~layer"

Utilisez ces variables afin de détecter la collision avec le layer Enemy.

→ Comment créer un Raycast (la direction sera "**out hit**")

→ Comment utiliser le type Ray (pensez à Input)

```
private IEnumerator Fire()
{
    //Set boolean to true (coroutine not free)
    // Play shoot sound
    //Play animation of weapon_active
    //shoot in game
   ray = child_camera.ScreenPointToRay (Input.mousePosition);
    if (Physics.Raycast (ray, out hit, Mathf.Infinity, layer))
        //hit is the killed enemy GameObject reference

    yield return new WaitForSeconds ((weapon_active.name == "Gun") ? 0.75f : 0.1f);
    //Set boolean to false (coroutine free)
}
```

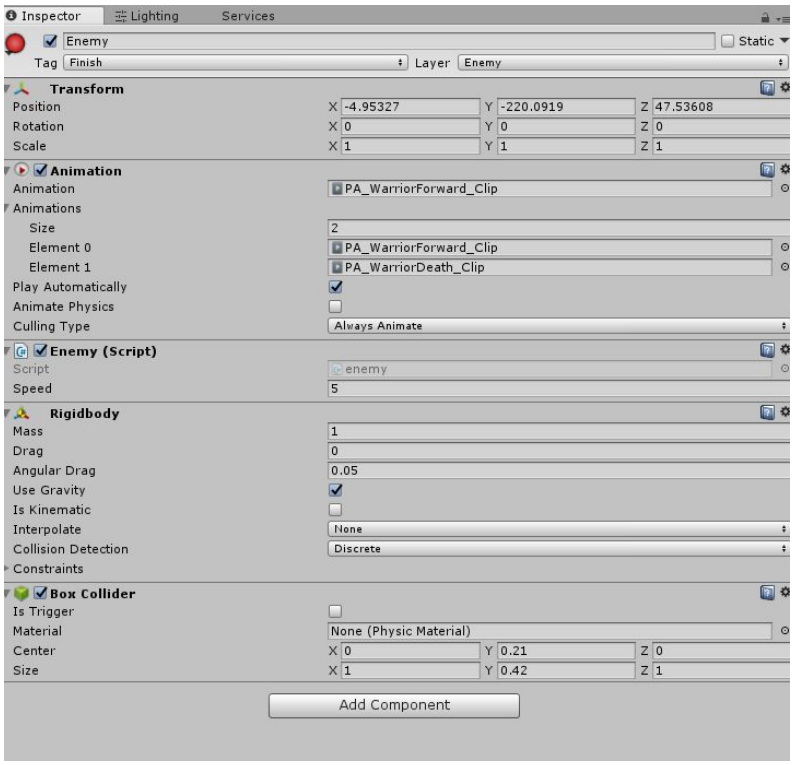
Grâce à ScreenPointToRay, on obtient la ligne partant de la caméra jusqu'à la position actuelle de la souri, c'est à dire le parfait milieu de l'écran. On vérifie ensuite la collision de cette ligne avec le layer. Concernant le "out hit": [modificateur de paramètre C#]

Le player est maintenant terminé, vous pouvez tirer, changer d'arme, etc... . Le script devrait ressembler à cela (le succès du Raycast sera traité plus tard):

```
public class Player : MonoBehaviour {
    public GameObject[] weapons;
    public AudioSource weapons_sound;

    private GameObject weapon_active;
    private bool can_shoot = true;

    private Camera child_camera;
    private RaycastHit  hit;
    private Ray ray;
    private int layer = 1 << 8;

    private void Start()
    {
        weapon_active = weapons [0];
        child_camera = gameObject.GetComponentInChildren<Camera> ();
    }

    private void Update()
    {
        if (Input.GetKeyDown (KeyCode.A)) {
            Switch_weapon ();
        }
        if (Input.GetKey (KeyCode.Mouse0) && can_shoot) {
            StartCoroutine(Fire ());
        }
    }

    private void Switch_weapon()
    {
        if (weapon_active.name == "Gun") {
            weapon_active = weapons [1];
            weapons [0].SetActive (false);
        } else if (weapon_active.name == "Gun2") {
            weapon_active = weapons [0];
            weapons [1].SetActive (false);
        }
        weapon_active.SetActive (true);
    }

    private IEnumerator Fire()
    {
        can_shoot = !can_shoot;
        weapons_sound.Play ();
        weapon_active.GetComponent<Animation> ().Play ();
        ray = child_camera.ScreenPointToRay (Input.mousePosition);
        if (Physics.Raycast (ray, out hit, Mathf.Infinity, layer))
            hit.collider.gameObject.GetComponent<enemy> ().Death ();
        yield return new WaitForSeconds ((weapon_active.name == "Gun") ? 0.75f : 0.1f);
        can_shoot = !can_shoot;
    }
}
```

- Les ennemis, animation, mouvements

Vous allez créer vos premiers ennemis, utilisez la prefab déjà préparée dans Done/Prefab/Enemy



Cette prefab contient déjà deux animations, déplacement et mort, l'animation de déplacement se joue automatiquement. Elle contient également un rigidbody afin de respecter la physique. Le component Box Collider permet de ne pas pouvoir la traverser, il représente la "hit box" 3D de l'ennemi.

Vous devez réaliser le script de l'ennemi.

- Variables

```
5 public class enemy : MonoBehaviour {
6     public float speed;
7
8     private GameObject target;
9     private Vector3 mvt;
0     private float step;
1     private Animation anim;
2     private bool alive = true;
3
```

speed est la vitesse de l'ennemi.
target, mvt et step vont servir au déplacement.
anim pour accéder à l'objet Animation
alive pour avoir un état de l'ennemi.

Dans ce cas précis, l'objet Animation est sur le même Objet que le script. On peut donc créer une référence à l'Animation de l'Objet directement dans le script.

$$anim = gameObject.GetComponent < Animation > ();$$

Utiliser gameObject revient à utiliser this. ***(On utilisera ici la référence à Animation uniquement pour la mort de l'ennemi)***

- *Fonction Move, Vector3, Transform*

Il faut donc faire bouger l'ennemi dans la direction du joueur. On a déjà parlé des layers, il existe aussi les tags. Créez un tag Player et changer le tag du FPSController en Player. De cette façon on peut "chercher" l'objet target avec un tag.

→ Récupérer un GameObject à l'aide de son tag

Une fois que vous avez trouvé votre objet, stockez sa position dans un Vector3 (mvt).

La class Vector3 contient une fonctionnalité très utile: MoveTowards. Elle permet de changer la position d'un objet afin qu'il se déplace vers un point donné.

→ Déplacer un Objet vers un point

Si vous laissez un ennemi dans la scène vous verrez que l'ennemi vous suit partout, cependant son orientation est toujours la même…

→ Rotation LookAt

En quelques lignes, vous avez un ennemi qui vous traque simplement.
Cette technique de suivi est simple mais à des défauts, les ennemis risquent fortement de se bloquer dans des coins si votre environnement est de type "close quarter". Pour remédier à cela il faudrait implémenter un A* :).

Il se peut que l'animation soit trop lente pour le déplacement de l'ennemi. Pour changer la vitesse d'une animation:

$$anim[string\ animation\ name].speed = (float);$$

Appelez votre fonction Move() dans Update(), ajoutez une condition pour bouger l'ennemi uniquement si il est en vie.

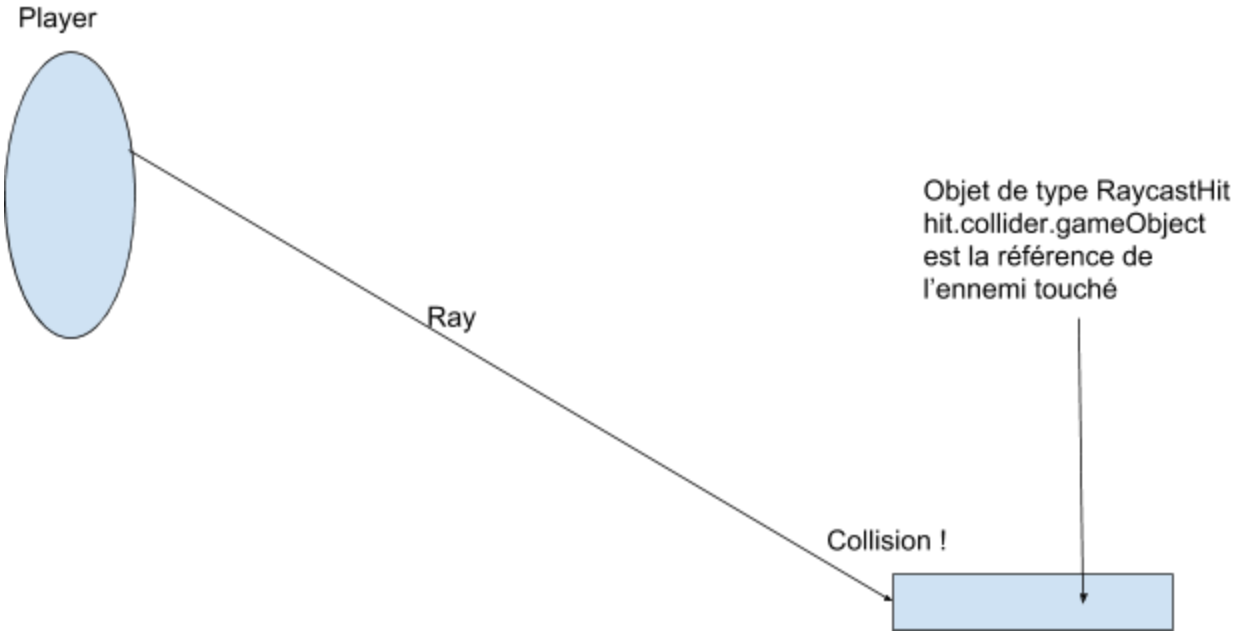
- Fonction Death(), Destroy, RaycastHit

Maintenant que les ennemis se déplacent, vous pouvez lier la détection de collision du Player pour tuer l'ennemi.

Créez une fonction Death dans la class enemy. Cette fonction va changer alive à false, jouer l'animation de mort, et Destroy le GameObject après un certain temps. **Cette fonction sera public**, de cette façon le script Player pourra y accéder.

→ Destroy un GameObject

Dans votre fonction de Raycasting, si la condition de collision est remplie, appelez la fonction Death de l'ennemi touché.



Pour appeler une fonction présente dans un autre script, on utilise l'Objet sur lequel est attaché ce script.

$$hit.collider.gameObject.GetComponent < class\ name > ().Function();$$

→ Fonction de Debug (le Debug permet notamment d'afficher les Ray)

Voici le script de l'ennemi :

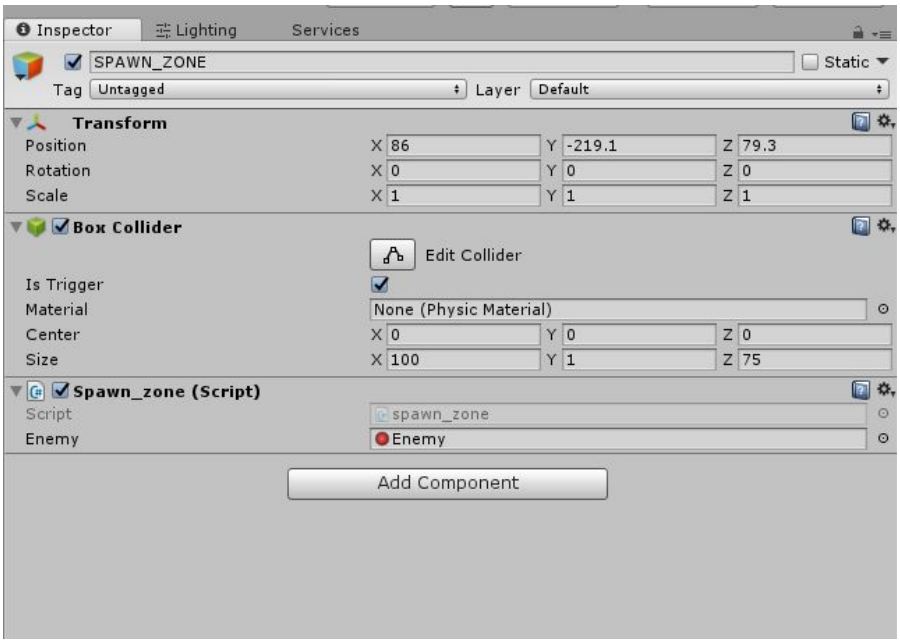
```
 5 public class enemy : MonoBehaviour {
 6     public float speed;
 7
 8     private GameObject target;
 9     private Vector3 mvt;
10     private float step;
11     private Animation anim;
12     private bool alive = true;
13
14     private void Start ()
15     {
16         anim = gameObject.GetComponent<Animation> ();
17         anim ["PA_WarriorForward_Clip"].speed = 2.0f;
18     }
19
20     private void Update ()
21     {
22         if (alive)
23             Move ();
24     }
25
26     private void Move()
27     {
28         target = GameObject.FindGameObjectWithTag ("Player");
29         mvt = target.transform.position;
30         step = speed * Time.deltaTime;
31         gameObject.transform.position = Vector3.MoveTowards (gameObject.transform.position, mvt, step);
32         gameObject.transform.LookAt (target.transform);
33     }
34
35     public void Death()
36     {
37         alive = false;
38         anim.Play ("PA_WarriorDeath_Clip");
39         Destroy (gameObject, 5.0f);
40     }
```

```
52         if (Physics.Raycast (ray, out hit, Mathf.Infinity, layer))
53             hit.collider.gameObject.GetComponent<enemy> ().Death ();
```

- Zone de Spawn, Box Collider Area, Instanciate, Quaternion

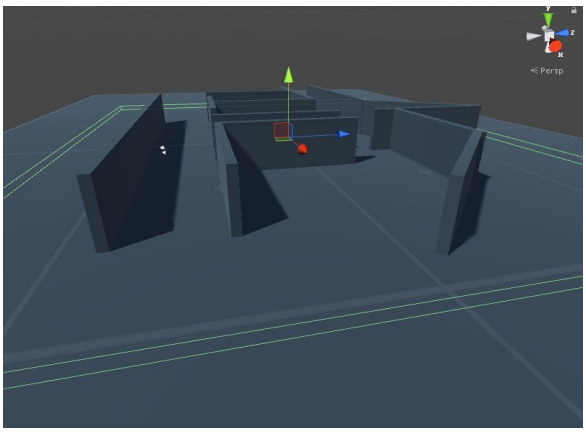
Pour créer la zone de spawn, créez un Objet vide dans la hiérarchie (Create -> Empty) et renommer le SPAWN_ZONE. Ajouter les Components comme ci-dessous.

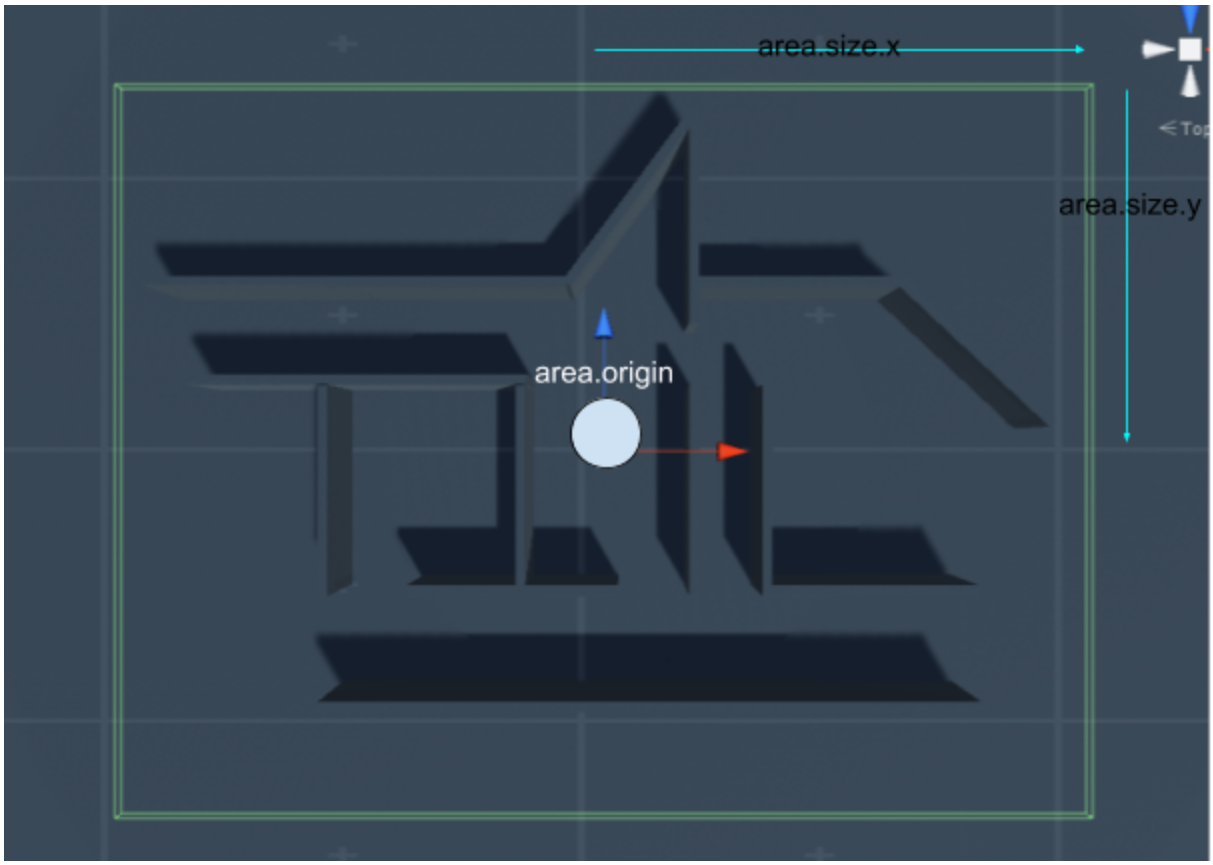
Vous avez peut être remarqué que la checkbox “Is Trigger” du component est cochée, alors qu’elle ne l’était pas sur l’ennemi. Le caractère trigger d’un Collider permet de savoir si oui ou non, l’accès à l’intérieur de la Box est accepté. Pour l’ennemi, on ne voulait pas pouvoir le traverser, ou que les ennemis se traversent mutuellement, on décoche donc “Is Trigger”. Cependant la zone de spawn va englober l’espace jouable, si le caractère trigger est désactivé, le joueur ne pourra pas bouger.



Le script Spawn_zone à besoin d’une référence de l’Objet Enemy. En effet pour Instancier un ennemi, on doit avoir une référence sur celui-ci.

→ Le component Box collider





Vous allez créer un vector3 à chaque instanciation de l'objet Enemy. Ce vector3 représentera la position du spawn. Pour cela, utilisez les tailles du component Box Collider (ci dessus, où area est un objet de type BoxCollider).

$$Vector3\ pos\ =\ new\ Vector3(x,\ y,\ z);$$

Afin d'ajouter de l'aléatoire, vous pouvez utiliser la class Random : Utiliser Random .

Cette position sera envoyée à une Coroutine. Elle instancie donc un (ou plusieurs) enemy à la position reçue, et attend un certain moment avant de retrouver une position.

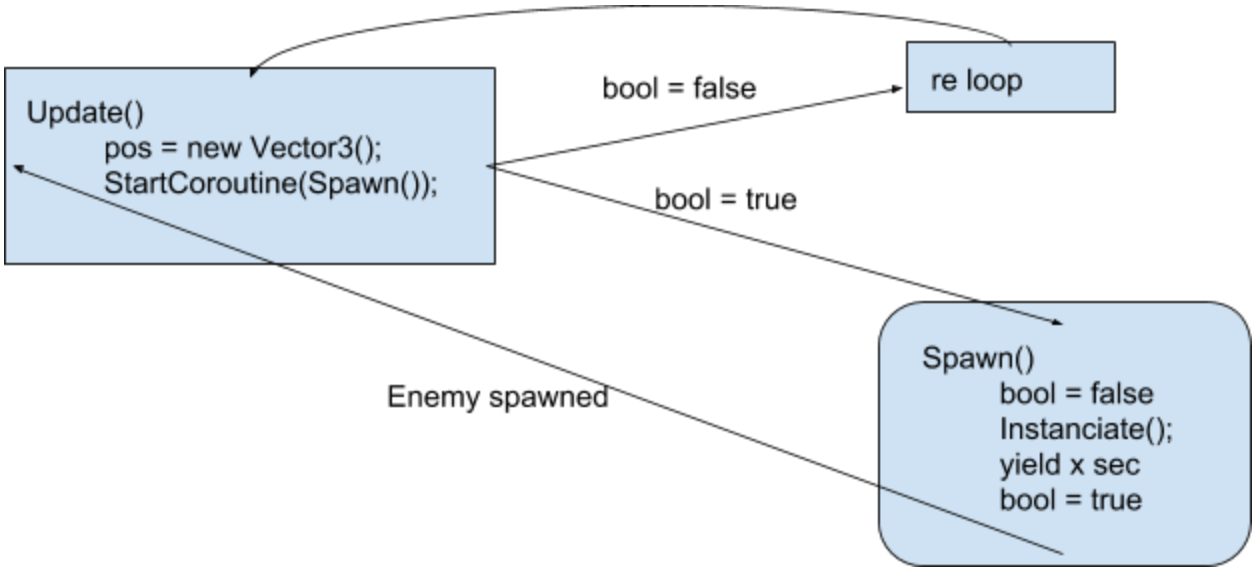
→ Instanciation d'un Objet

$$public\ static\ Object\ Instantiate(Object\ original,\ Vector3\ position,\ Quaternion\ rotation);$$

Le Quaternion représente la rotation d'un objet *(il utilise des nombres complexes)*.

Afin de garder une rotation par défaut, on envoie Quaternion.identity en troisième argument.

Déroulement du Spawn



Essayer de réaliser ce script sans regarder la réponse.

```
public class spawn_zone : MonoBehaviour {
    public GameObject enemy;

    private BoxCollider area;
    private Vector3 pos;
    private bool can_spawn = true;

    private void Start()
    {
        area = gameObject.GetComponent<BoxCollider> ();
    }

    private void Update()
    {
        if (can_spawn) {
            pos = new Vector3 (gameObject.transform.position.x + Random.Range (-area.size.x, area.size.x),
                gameObject.transform.position.y + Random.Range (-area.size.y, area.size.y),
                gameObject.transform.position.z + area.center.z);
            StartCoroutine (Spawn (pos));
        }
    }

    private IEnumerator Spawn(Vector3 spawn_pos)
    {
        int r = Random.Range (1, 4);

        can_spawn = !can_spawn;
        for (int i = 0; i < r; i++) {
            Instantiate (enemy, spawn_pos, Quaternion.identity);
            spawn_pos.x += (i * enemy.transform.lossyScale.x);
        }
        yield return new WaitForSeconds (3.0f);
        can_spawn = !can_spawn;
    }
}
```

Bravo si vous êtes arrivé ici! Votre jeu est pratiquement terminé, il n'y a plus qu'à rajouter un GameOver.

- Dernière étape, OnCollision, type Collision, Canvas

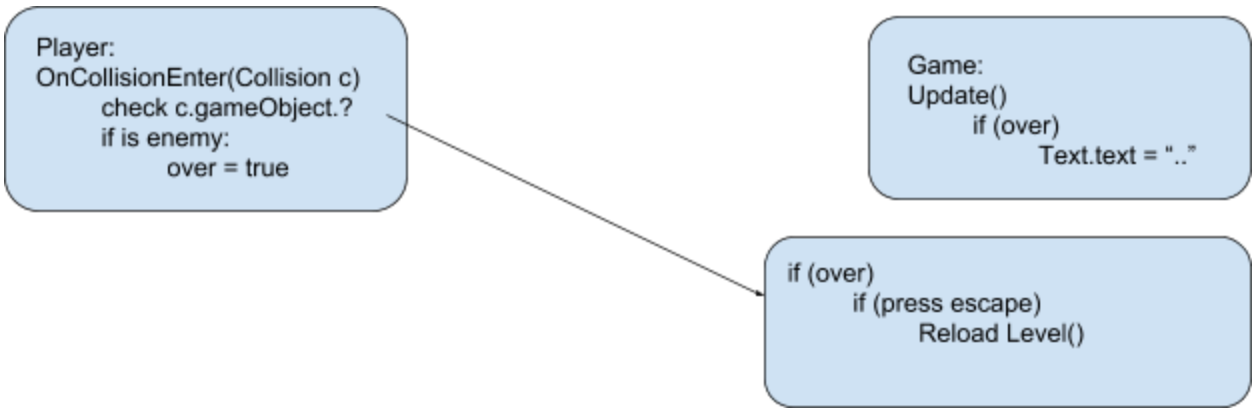
Vous allez devoir créer un objet Text, pour cela: Create->UI->Text. Cela va créer un Objet Canvas, avec pour enfant votre objet Text. Vous pouvez utiliser son component Rect Transform afin de le positionner sur l'écran de jeu.

→ Toutes les infos sur Text

Vous avez également besoin d'une variable global, par exemple un boolean over. Lorsque le Player entrera en collision avec un ennemi, cette variable passera à true, affichant alors le GameOver.

→ Comment détecter les collisions

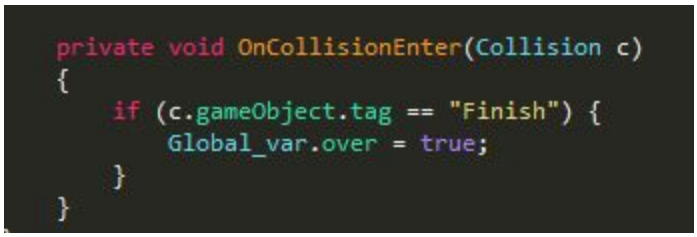
Pour résumé:



→ Charger une Scène

Attention lorsque vous utilisez SceneManager.LoadScene('toto') Unity va chercher la scène toto dans les scènes "build", pour ajouter une scène dans le "build": File->build Settings->Add open Scene.

Au final, il faut rajouter cette fonction dans le Player:



La class Global_var est une class static contenant la variable globale.

→ Global variables in C#

Vous avez terminé ce workshop, Félicitation !

Si vous êtes intéressé par le Post Processing (traitement de l'image), rendez vous à la page suivante.

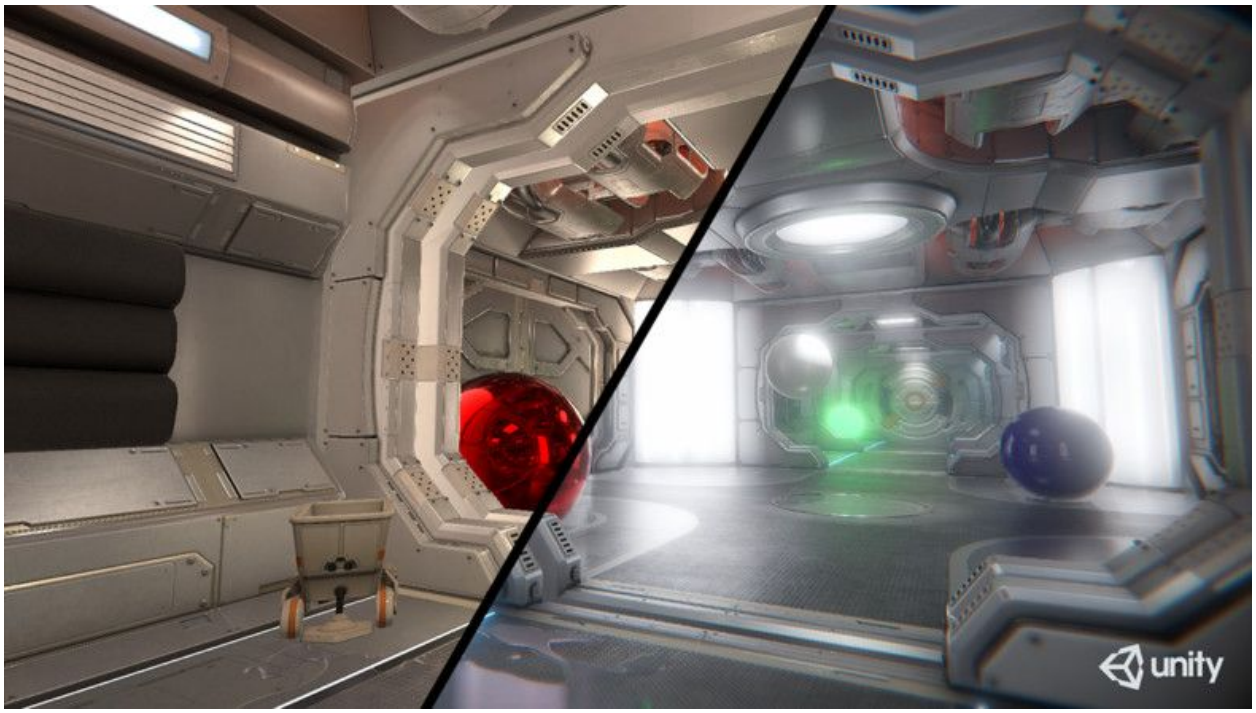
- Post Processing

Grâce au Post Processing, vous pouvez nettement embellir votre jeu.

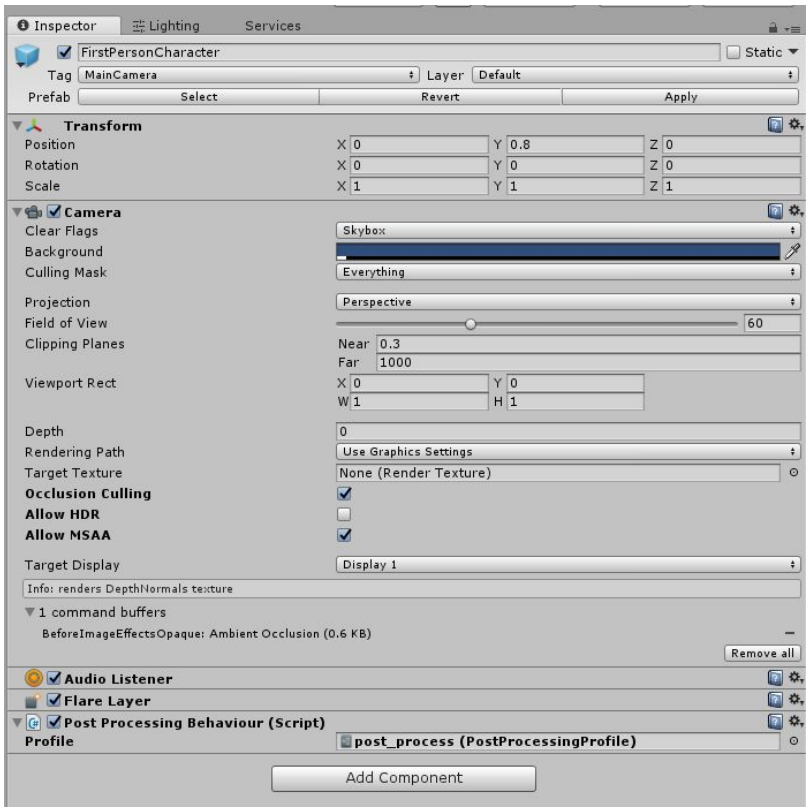
Le Post Processing permet d'appliquer des filtres ou des traitements d'images au buffer de la caméra avant l'affichage.

→ Voici les filtres disponibles

Les plus courant sont le FXAA (AntiAliasing), permettant de corriger les bordures des Objets; L'occlusion Ambiante, permettant de créer des zones d'ombres dans les coins; Le Bloom, permettant d'accentuer l'éblouissement des émissions de lumières; Le Color Grading, qui permet de modifier les couleurs de l'image; Et pour finir, la Vignette, qui permet de donner un effet "vue par les yeux" à l'image.



Pour l'intégrer à votre jeux, vous devrez télécharger la PostProcessing Stack sur l'Asset Store. [PostProcessing Stack]



Vous devez ajouter le Script "Post Processing Behaviour" à votre caméra.

Le profile de post processing se fait dans le projet via Create->Post Processing Profile.

Il suffit de jouer avec les filtres pour obtenir un résultat.