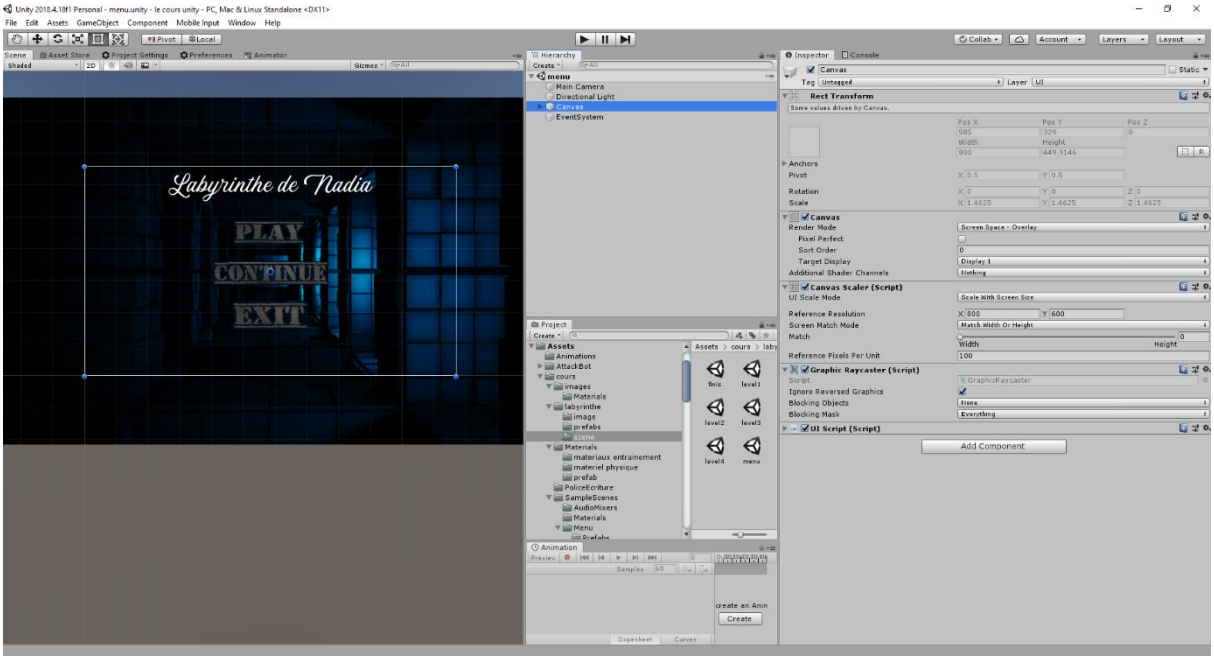
Les scripts Unity les plus importants

Sommaire :

Ce qu'il faut savoir : chaque script est associé à ce qu'on appelle un *gameObject*, un script peut être appelé dans un autre script. Tous les scripts n'ont pas été renseignés pour éviter les scripts qui contiennent la même chose que d'autre à quelques détails près.

UIScript :



Le script correspond au menu du jeu, il est associé à l'objet Canvas qui est en réalité une interface utilisateur, c'est une scène en 2D. L'interface utilisateur correspond au Canvas sur Unity.

```
public class UIScript : MonoBehaviour
{
    0 références
    public void Start()
    {
        Cursor.visible = true;
        Cursor.lockState = CursorLockMode.None;
    }
    0 références
```

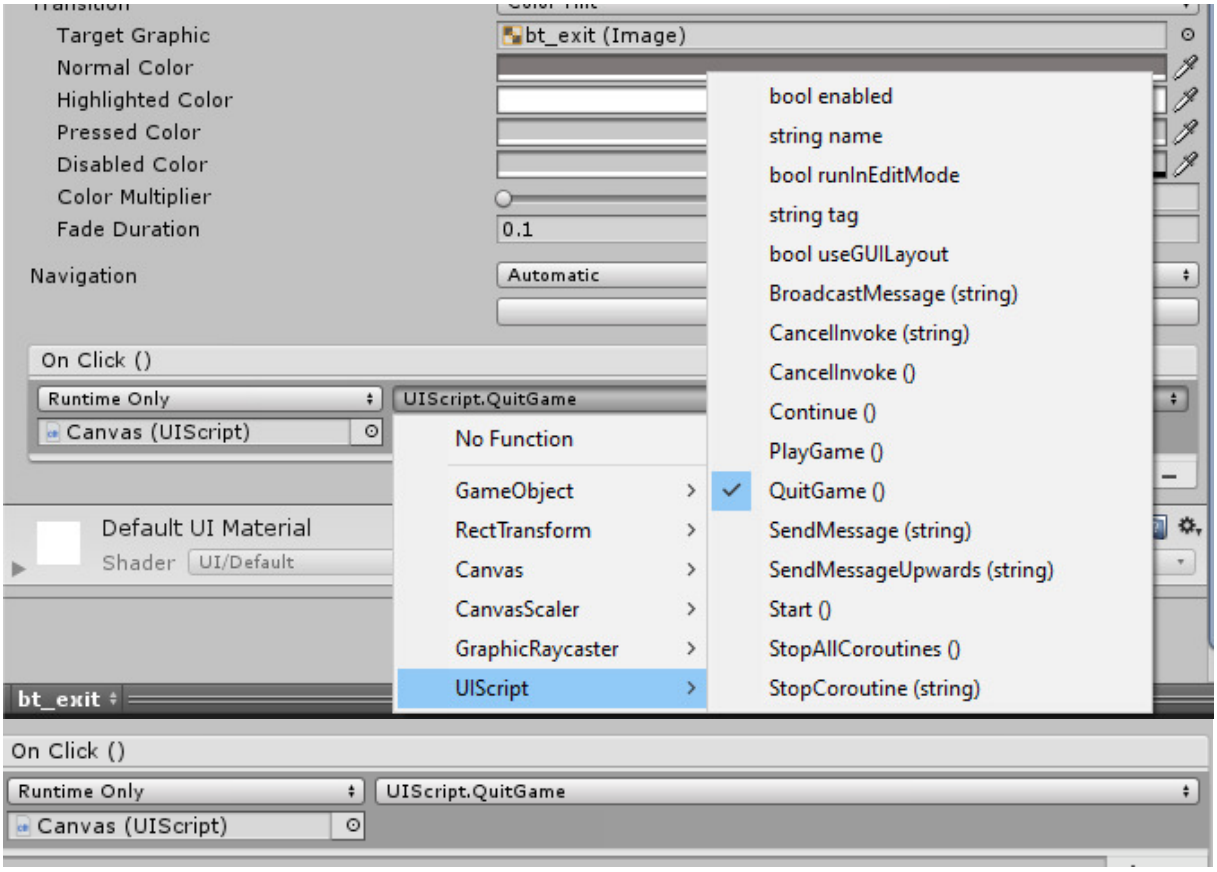
Public void Start est une fonction propre à Unity qui s'exécute automatiquement au lancement de la scène.

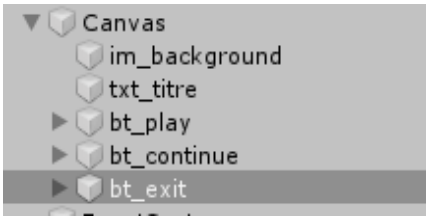
Cette fonction sert ici à afficher le curseur. Le souci des jeux à la première personne, est que, quand on est dans en mode FirstPersonn, le curseur disparait, et si l'on retourne ensuite dans le menu il n'y a donc plus de curseur.

La première ligne sert évidemment à afficher le curseur mais il est par défaut bloqué au milieu. D'où l'utilité de la deuxième ligne qui permet de le débloquer.

```
public void QuitGame()
{
    Application.Quit();
}
```

Cette fonction permet de quitter le jeu via le bouton Exit. Les boutons d'une interface utilisateur sont très particuliers car pour les faire fonctionner il faut leur donner une fonction.

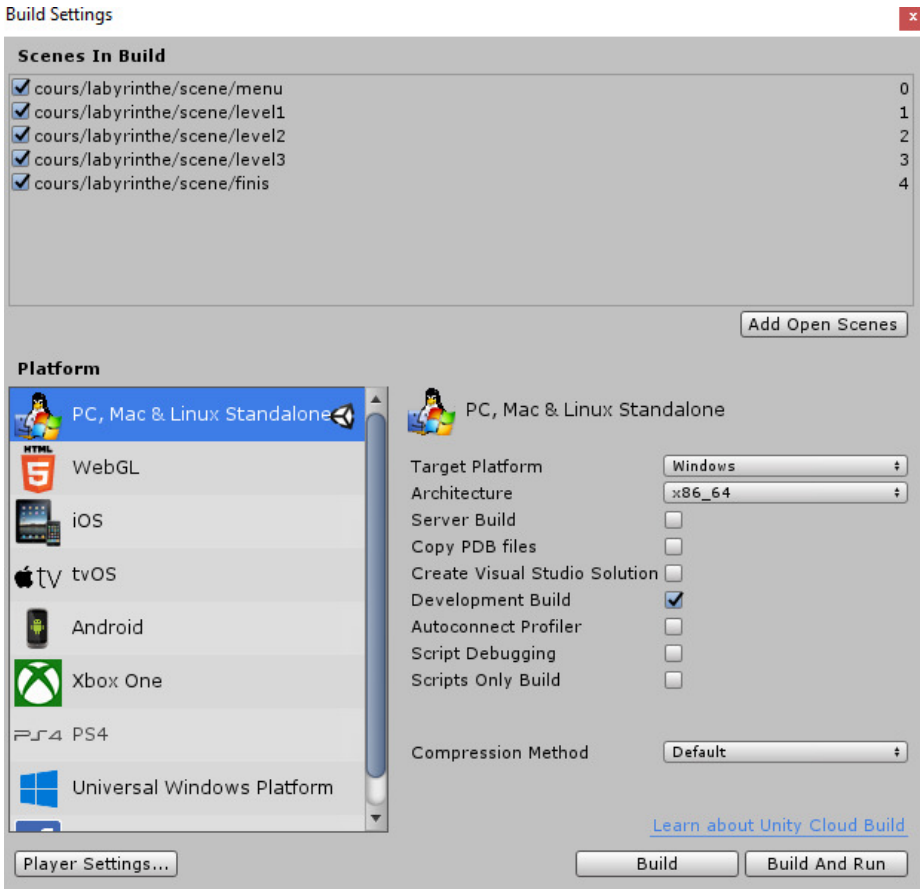




Comme c'est l'objet parent *Canvas* qui a le script, on associe le script au bouton donc le parent (l'image au-dessus) correspond à *l'inspector* de *bt_exit*). Ensuite on choisit une fonction à attribuer au bouton : ici, la fonction de notre script *QuitGame()*.

```
0 références
public void PlayGame()
{
    SceneManager.LoadScene(1);
}
```

SceneManager permet d'avoir la main sur toutes les scènes.



Voici toutes les scènes de la première version du jeu. On voit que chaque scène a un nom (« menu », « level1 » …) et un index. La fonction *PlayGame* se situera sur le bouton *play*, c'est-à-dire quand on joue mais en partant du premier niveau. On va donc charger la scène qui contient l'index 1, le level1.

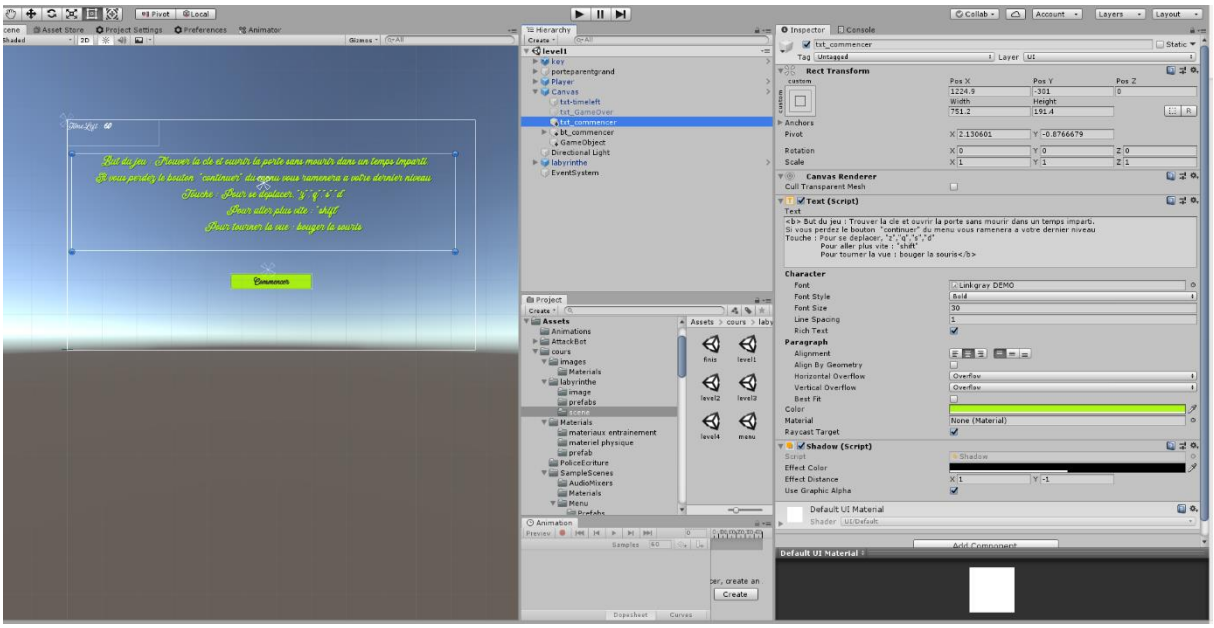
```
public void Continue()
{
    int levelToLoad = PlayerPrefs.GetInt("DernierNiveau");

    if(levelToLoad>1)
    {
        SceneManager.LoadScene(levelToLoad);
    }
    else
    {
        SceneManager.LoadScene(1);
    }
}
```

Cette fonction servira pour le bouton *Continue*. La variable *LevelToLoad* va nous permettre de récupérer le dernier niveau que le joueur a réussi à atteindre (voir *AfterBurnerScript OnTriggerEnter*). *PlayerPerfs* nous permet ici de récupérer les statistiques de notre joueur grâce à des get et une clé.

On fait ensuite un test, si le dernier niveau que l'on a atteint est supérieur à l'index 1 (donc le level1) on chargera la scène de ce dernier niveau. Mais si il est inférieur ou égal à 1 (c'est-à-dire que notre joueur n'a jamais joué ou n'a pas réussi le level1) on l'emmènera au niveau 1.

UiCommencer :



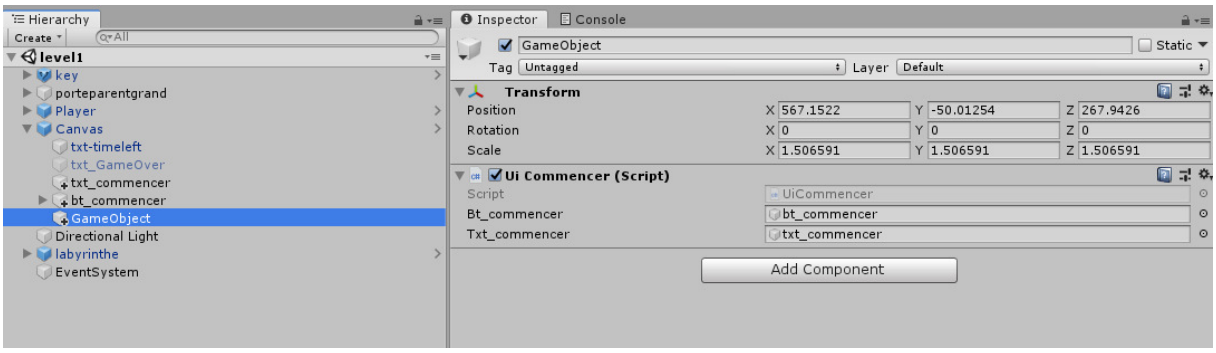
Le script correspond à un des scripts de l'interface utilisateur, celui qui permet de voir le texte et le bouton. Le script se situe sur un objet du nom de *GameObject* qui est un objet invisible, qui sert par exemple à créer des points de collision ou à faire des lien de parenté ou des Prefab.

```
[SerializeField]
GameObject bt_commencer;

[SerializeField]
GameObject txt_commencer;
```

Les variables sont déclarées en *SerializeField* pour que l'on puisse avoir la main dessus en dehors du Script mais aussi que l'on puisse attribuer des objets à ces variables sans faire de :

GameObject .Find().GetComponnent(script)



On peut voir que sur mes variables *SerializeField,* j'y ai glissé le texte et le bouton pour pouvoir les utiliser.

```
0 references
public void Start()
{
    Cursor.visible = true;
    Cursor.lockState = CursorLockMode.None;
    GameObject.Find("Player").GetComponent<FirstPersonController>().enabled = false;
    GameObject.Find("Canvas").GetComponent<CountDownScript>().enabled = false;
}
```

Le but de cette fonction ici est d'afficher encore une fois le curseur comme dans le menu, mais aussi et surtout de mettre le jeu en pause.

GameObject.Find('nom') : permet de trouver un objet par son nom

GetComponent<> : permet de récupérer son script.

Dans la troisième ligne je récupère le script qui permet au joueur de bouger et je le rends inactif. Ensuite je récupère le script de mon interface utilisateur qui permet **au TimeLeft de descendre**.

Si cette partie n'était pas réalisée, le joueur pourrait se déplacer alors qu'il est censé lire les règles du jeu, et le compteur descendrait : il mourrait à cause du temps écoulé pendant qu'il lit.

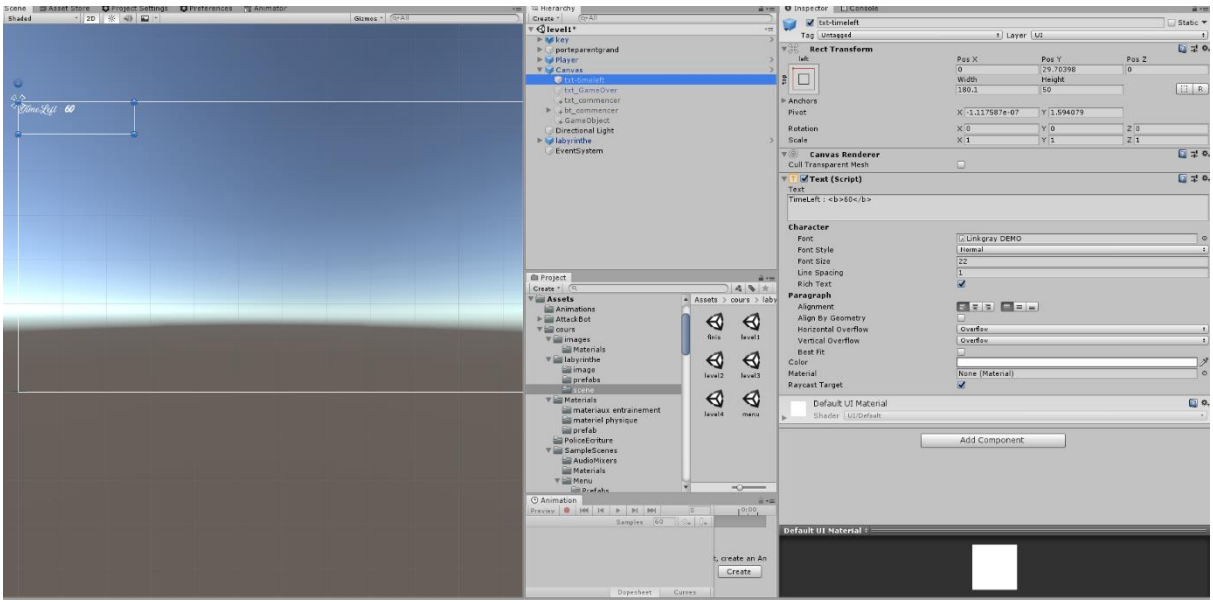
```
public void Commencer()
{
    GameObject.Find("Player").GetComponent<FirstPersonController>().enabled = true;
    GameObject.Find("Canvas"). GetComponent<CountDownScript>().enabled = true;
    txt_commencer.SetActive(false);
    bt_commencer.SetActive(false);
    Cursor.visible = false;
}
```

Cette fonction est attribuée au bouton **commencer** de notre interface. Quand le joueur appuiera, les deux scripts désactivés (le joueur et le compteur) seront réactivés, cependant, le bouton, le texte et le curseur disparaitront.

txt_commencer.SetActive(false) ;

On a attribué un GameObject à la variable *txt_commencer* ; on peut donc modifier les attributs : savoir si le GameObject de cette variable doit être activé ou non.

CountDownScript :



Ce script va nous permettre de mettre un compteur à notre jeu que le joueur verra. Et il va nous permettre de mettre en GameOver le joueur si le temps est écoulé.

```
[SerializeField]
private int startCountDown = 60;

[SerializeField]
Text txt_CountDown;
```

On met ici en variable *serialize* une variable *startCountDown* qui nous permettra de changer la valeur de compteur en dehors du script de façon définitive mais de garder 60 secondes pour les autres niveaux.

Dans ce script on aura besoin de modifier le compteur que le joueur verra ; on doit donc relier le texte de l'interface utilisateur au script pour modifier le message.

```
0 references
void Start()
{
    txt_CountDown.text = "TimeLeft : " + "<b>" + startCountDown + "</b>";
    StartCoroutine(Pause());
}
```

Au démarrage on modifie le texte pour qu'il commence au temps demandé et qu'il ne marque pas par exemple sur un niveau à 70 secondes : 60 ;69 ;68….

On appelle ensuite notre prochaine fonction qui permettra de lancer le compteur, StartCoroutine permet d'appeler une fonction IEnumerator pour jouer sur les temps de pause du script.

```
IEnumerator Pause()
{
    while (startCountDown > 0)
    {
        yield return new WaitForSeconds(1f);
        startCountDown--;
        txt_CountDown.text = "TimeLeft : " + "<b>" + startCountDown + "</b>";
    }
    GameObject.Find("Player").GetComponent<FirstPersonController>().GameOver();
    GameObject.Find("Player").GetComponent<FirstPersonController>().enabled = false;
}
```

Dans notre *while* qui permet de faire fonctionner le compteur, on a :

yield return new WaitForSecond(1f) ;

Cette fonction met la fonction en pause pendant *1f* ici donc une seconde. (Pas le script car si on a une ligne de *debug* par exemple dans notre fonction *Start* et que cette ligne se situe en dessous de l'appel de fonction, la ligne s'exécutera directement)

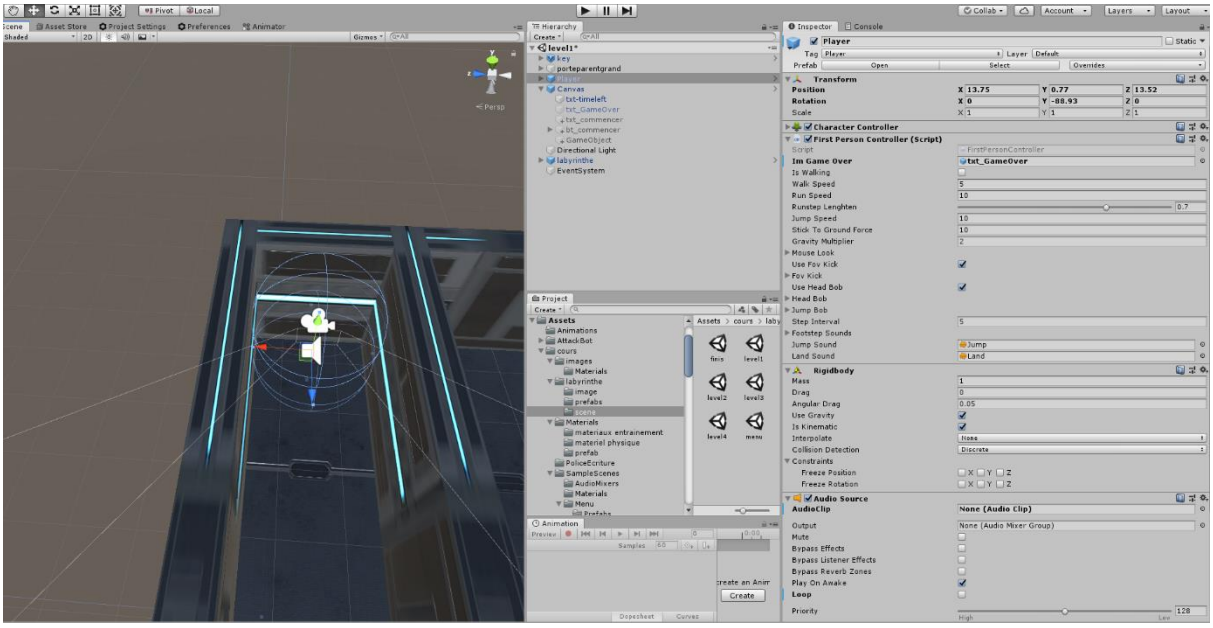
Ensuite on enlève 1 à notre fonction startCountDown et il ne nous reste plus qu'à modifier le texte pour qu'il actualise le compteur.

Quand le compteur est écoulé (en dehors du *while*) on appelle la fonction GameOver.

Dans notre script du joueur on a une fonction Player (voit *FirstpersonController GameOver*();) qui nous permettra de ramener le joueur au menu.

On désactive ensuite le script du joueur après avoir lancé le *GameOver* pour éviter qu'il puisse se déplacer (si par exemple il a perdu à 1 seconde de la porte on le bloque pour ne pas qu'il puisse passer et donc gagner en ayant « triché »).

FirstPersonController :



Par souci de comptabilité avec VisualStudio et mon ordinateur pendant longtemps, le script du joueur ne m'appartient pas, mais j'ai dû y apporter quelque modification que je vais donc présenter.

```
[SerializeField]
GameObject ImGameOver;
```

Dans notre interface utilisateur un message *GameOver* est caché pour éviter que le joueur ne le voie pendant qu'il joue. Comme le *GameObject* est désactivé de base on ne peut pas le modifier sauf si on le met manuellement dans une variable, c'est pour cela qu'on utilise ici une variable *serialize.*

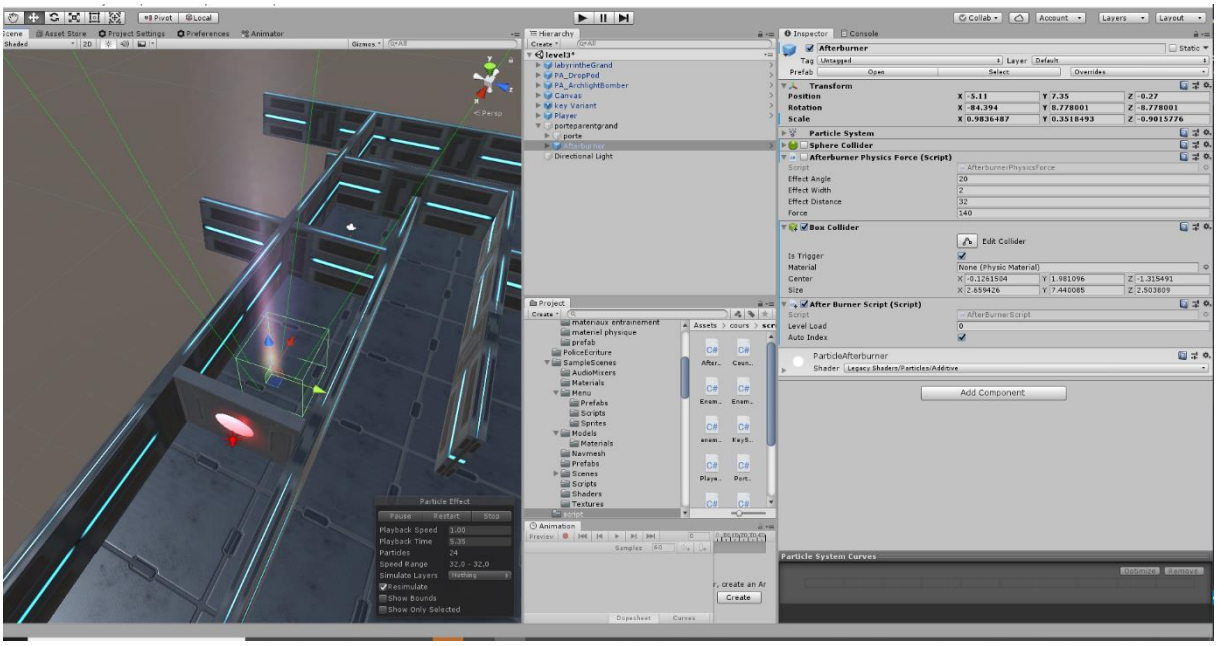
```
4 références
public void GameOver()
{
    ImGameOver.SetActive(true);
    StartCoroutine(LoadMenu());
}
```

Quand la fonction est appelée, le joueur voit le message caché apparaitre et notre fonction *LoadMenu* est appelée avec *StartCoroutine* pour permettre de mettre la fonction en pause le temps de l'affichage du *GameOver*

```
IEnumerator LoadMenu()
{
    yield return new WaitForSeconds(2f);
    SceneManager.LoadScene(0);
}
```

Cette fonction permet de mettre en pause cette même fonction pendant deux secondes le temps de l'affichage du GameOver situé dans la précédente fonction, puis de charger le menu qui se trouve à l'index 0.

AfterBurnerScript :

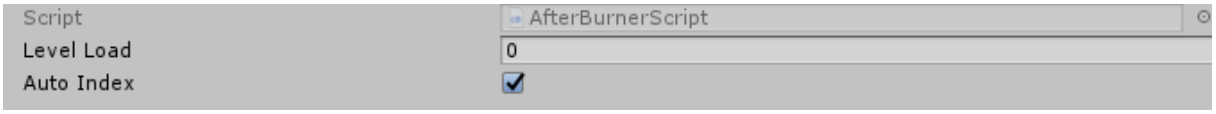


Le script correspond à l'objet sélectionné, il se situe donc dans tous les niveaux du jeu (le cube vert correspond à la zone de collision (*Box collider*) ici la zone de collision est en « *is Trigger* » ce qui signifie qu'on peut passer à travers)

```
0 references
public class AfterBurnerScript : MonoBehaviour
{
    [SerializeField]
     int LevelLoad;

    [SerializeField]
    bool autoIndex = true;
```

Les variables sont déclarées en *SerializeField* pour qu'on puisse avoir la main dessus en dehors du Script (Tout changement opéré en dehors du script est définitif par exemple on décide d'être au niveau 5, si on met 6 le niveau deviendra 6)



On aperçoit les données en *serializeField* dans l'*inspector* de l'objet

```
0 références
public void Start()
{
    if(autoIndex)
    {
        LevelLoad = SceneManager.GetActiveScene().buildIndex + 1;
    }

}
0 références
```

Public void Start est une fonction propre à Unity qui s'exécute automatiquement au lancement de la scène.

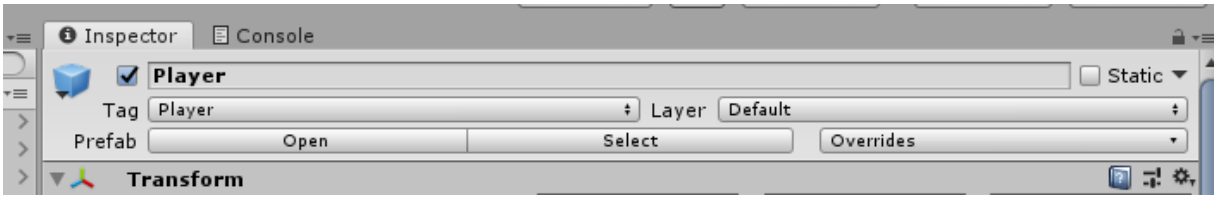
Le but de notre animation est de faire en sorte que quand on rentre dans l'animation, on soit au niveau suivant. Donc on prend l'index de la scène en cours et on lui met l'index plus 1. (Les scènes sont mises dans un ordre prédéfini et on décide soit de les appeler par leur nom, soit par leur Index)

Pourquoi *autoIndex* ? le souci ici c'est que si on souhaite par exemple pendant notre développement du jeu de passer du niveau 1 au niveau 50, donc qu'on change à la main la variable *LevelLoad,* le jeu va automatiquement remettre l'index 2 comme niveau suivant. C'est pour cela que l'on utilise un *boolean* : s'il est à *false* ce qu'il y a dans le *if* ne se fera pas.

```
0 references
private void OnTriggerEnter(Collider other)
{
    if (other.tag == "Player")
    {
        PlayerPrefs.SetInt("DernierNiveau", LevelLoad);
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
    }
}
```

OnTriggerEnter est une fonction spécifique à Unity, ici cette fonction permet de faire une action quand on entre dans la *box Collider* de notre animation. S'il y avait une collision on utiliserait *OnCollisionEnter* par exemple. Il existe 3 types de *OnTrigger, OnTriggerEnter* qui agissent quand on entre dans le *box collider*, *OnTriggerStay* agit quand on reste dans le collider. *OnTriggerExit* agit quand on sort du *box collider* (ces fonctions existent aussi avec 2D à la fin de leur nom pour faire des jeux en 2D).

Le paramètre *other* correspond à l'objet qui entre dans le *box collider.* Sur chaque gameObject on peut ajouter un tag ce qui est pratique pour les collisions par exemple.



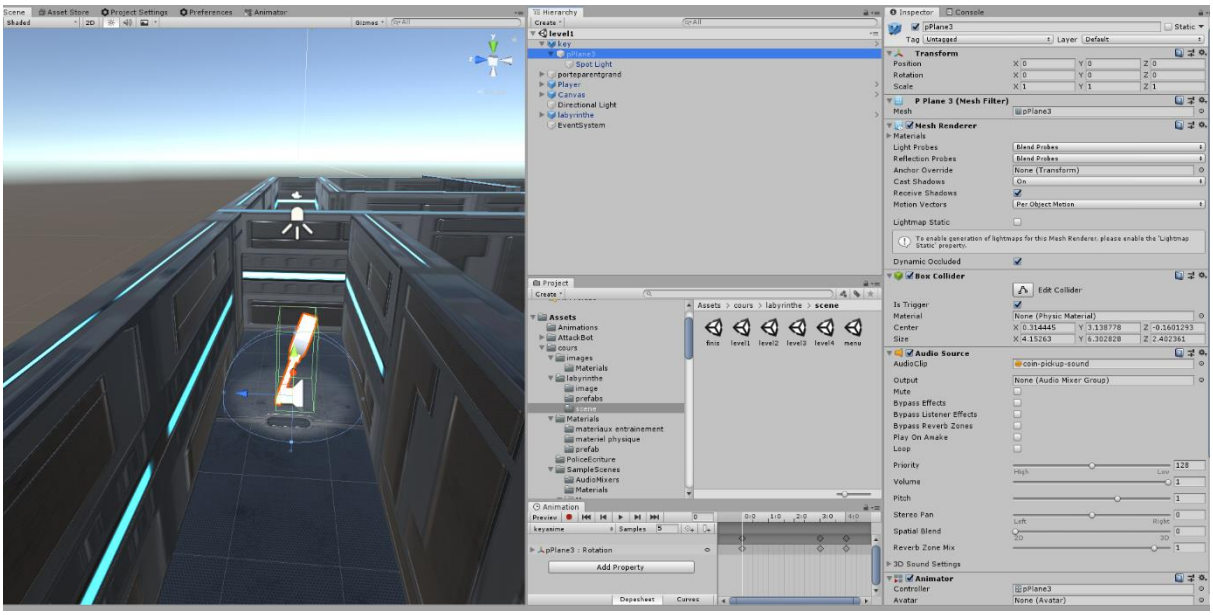
Ici on peut voir que le *gameObject* de mon joueur est *Player* mais que son *tag* aussi.

Donc quand le *box collider* de mon joueur rentrera en contact avec celui de l'animation, on pourra charger la scène suivante.

PlayerPrefs est, elle aussi, une fonction spécifique à *Unity*. C'est dans cette fonction que l'on va stocker les statistiques de notre joueur. Ici pour le bouton continuer de notre Menu on va enregistrer le dernier niveau où il se trouve. Donc quand le joueur franchit l'animation, cela veut dire que le dernier niveau auquel il pourra continuer est le niveau suivant. On utilise une clé pour pouvoir faire un *PlayerPrefs.getInt("clé")* plus tard et récupérer la valeur enregistrée.

La fonction *LoadScene* de *SceneManager* permet de charger la scène qu'on lui donne en paramètre.

KeyScript :



Le script correspond au script de la clé, pour que le joueur puisse la « récupérer ».

```
0 références
private void OnTriggerEnter(Collider other)
{
    if (other.tag == "Player")
    {
        GetComponent<AudioSource>().Play();
        GameObject.Find("porteparentgrand").GetComponent<PorteScript>().CanOpen = true;
        GetComponent<MeshRenderer>().enabled = false;



        Destroy(transform.parent.gameObject,0.3f);
    }
}
```

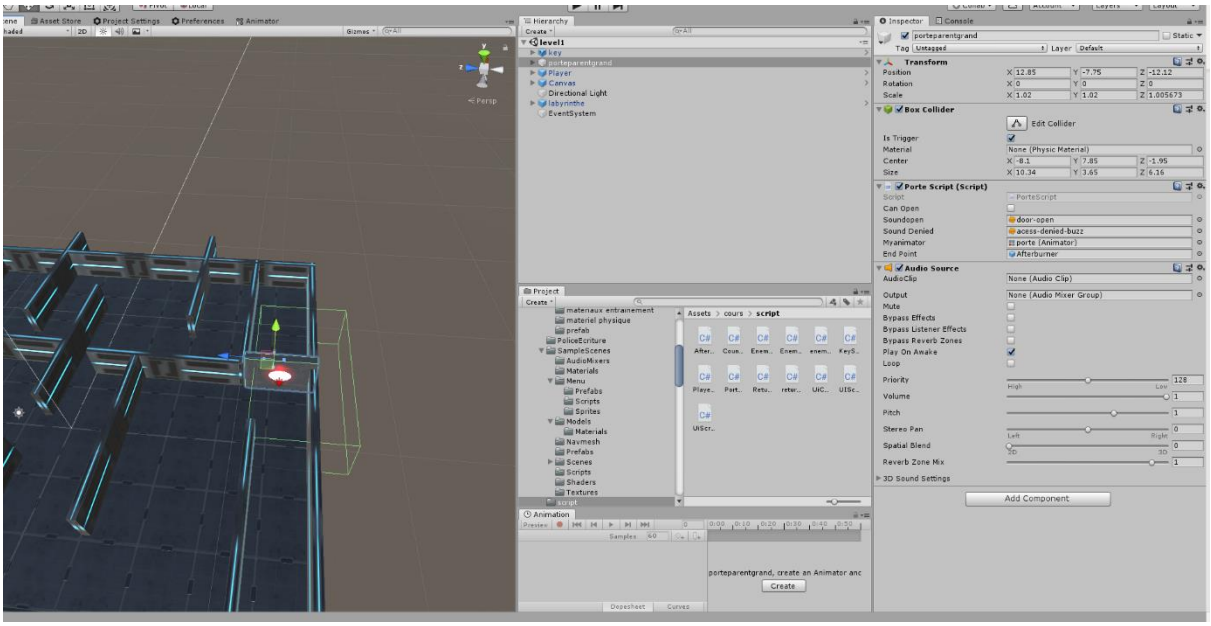
Ici on repart sur le système de collision, quand le joueur rentre dans le *box collider* de la clé. On a ajouté sur la clé un audio, mais cet audio ne doit être joué que quand on récupère la clé ; c'est pour ça qu'on l'active au moment où le joueur entre dans le *box collider*. Comme *l'AudioSource* est sur le même objet que le script, on n'a pas besoin de le chercher.

CanOpen est une variable *boolean* à *false* par défaut qui permettra d'ouvrir la porte (voir *PorteScript OnTriggerEnter*).

Le *MeshRenderer* correspond à l'objet 3D de la clé ; quand le joueur a pris la clé, elle doit être effacé du jeu. On l'efface donc sans le supprimer directement pour que le son de récupération de la clé ait le temps de se jouer.

C'est ensuite que l'on détruit la clé en détruisant le parent (le *prefab* complet), cette clé est détruite au bout de 0,3 seconde.

PorteScript :



Le script est associé à la porte.

```
public bool CanOpen = false;
[SerializeField]
AudioClip soundopen, soundDenied;

[SerializeField]
Animator myanimator;

[SerializeField]
GameObject endPoint;

bool couleur=false;

private AudioSource myAudioSource;
```

CanOpen est en public pour être accessible à tous les scripts, on le met à *true* quand on récupére la clé.

Sur la porte deux sons vont être joués quand on passe dans le *box collider* : le son de la porte qui s'ouvre et celui de la porte qui refuse de s'ouvrir. Ils sont en *serialize* pour que l'on puisse les ajouter à la main.

On récupère de la même manière que les sons, l'animation de la porte qui s'ouvre car elle ne doit pas s'ouvrir au tout début du niveau mais seulement avec certaines conditions.

Le *GameObject* correspond à l'animation qui permet de changer de niveau (voir *AfterBurnerScript*), et pour que le jeu prenne moins de ressource, l'animation n'est activée qu'à l'ouverture de la porte.

La porte a une lumière rouge de base, qui passe ensuite au vert. Comme ce changement de couleur va être généré dans une fonction qui est appelée toutes les microsecondes, on a besoin que l'action ne se fasse qu'une fois au lieu de tourner à l'infini.

On a récupéré deux audios, mais il faut les mettre dans *l'AudioSource* de la porte. On a donc besoin d'une variable *AudioSource*

```
private void Awake()
{
    myAudioSource = GetComponent<AudioSource>();
}
```

Awake est une fonction qui se déroule avant le start. C'est une fonction *Unity*, et c'est dans cette fonction que l'on définit notre AudioSource comme *l'AudioSource* de l'objet (plus précisément, on prend *l'AudioSource* de la porte).

```
public void FixedUpdate()
{
    if(CanOpen&&!couleur)
    {
        GameObject.Find("Point Light").GetComponent<Light>().color = Color.green;
    }
}
```

La fonction *FixedUpdate* appartient aussi à *Unity*. C'est une variable qui s'exécute à un intervalle très court et régulier (plus rapide qu'une seconde).

C'est dans cette fonction que nous allons changer la couleur de la lumière de la porte. Donc si on peut ouvrir la porte et que la couleur est à *false*, on récupère l'objet lumière (*Point Light*) et on lui attribue la couleur verte.

```
private void OnTriggerEnter(Collider other)
{
    //si le tag de l'objet qui rentre et player dansle collider
    if(other.tag=="Player" && CanOpen)
    {
        myanimator.enabled = true;
        myAudioSource.PlayOneShot(soundopen);
        endPoint.SetActive(true);
    }
    else
    {
        myAudioSource.PlayOneShot(soundDenied);
    }
}
```

On réutilise *OnTriggerEnter* pour savoir quand le joueur passe à côté de la porte. On effectue ensuite un test, si on a la clé, on active l'animation d'ouverture, on active le son d'ouverture et on active l'animation qui va nous permettre d'aller au niveau suivant.

Sinon on active seulement le son de refus d'ouverture.