Master Bio-Informatique

TDs : Prise en main d'Unity3D et création d'un moteur audio

Raphaël Marczak
15/01/2019

Dans ces TDs, nous allons prendre en main Unity3D sur un exemple simple (en 2D) :

- un personnage se déplace dans les quatre directions (initiation à la caméra, à la classe *gameobject*, à l'utilisation des sprites, aux commandes clavier, à la visibilité des variables dans l'inspecteur et à l'ajout de scripts C#)
- il peut récupérer des pièces et afficher un score sur l'écran (initiation au principe de collision, à l'échange de variables entre gameobjects, aux tags et à la GUI)

Ensuite, nous réaliserons un moteur audio simplifié, sous la forme d'un singleton :

- lecture de sons et de musiques (initiation à *AudioSource* et *AudioClip*)
- modification des paramètres sonores
- liens entre le personnage créé plus haut, et le moteur audio

Enfin, vous pourrez être plus créatifs en ajoutant :

- Un système d'apparition automatique d'objets à collecter
- Un personnage qui se déplace tout seul

---

1 – Prise en main d'Unity3D

Généralités sur Unity3D

- Moteur de jeu, de plus en plus utilisé par les développeurs de jeux vidéo (entreprises ou indépendants, etc.)
- Permet de lier facilement des objets graphiques avec des comportements (gameplay, animations, sons, etc.)
- Permet d'écrire des scripts en C#, Javascript et Boo, pour personnaliser les comportements.
- Gratuit depuis mars 2015 (avec quelques limitations), et permet de compiler pour PC, Mac, Linux, Web, Playstation, Nintendo, Xbox...)
- Attention... très (très !) facile de faire du code sale... Le paradigme de programmation est plus proche de « fonctionnalités/scripts » que de « classes/objets », même si le code est basé sur du C#

Premier jeu

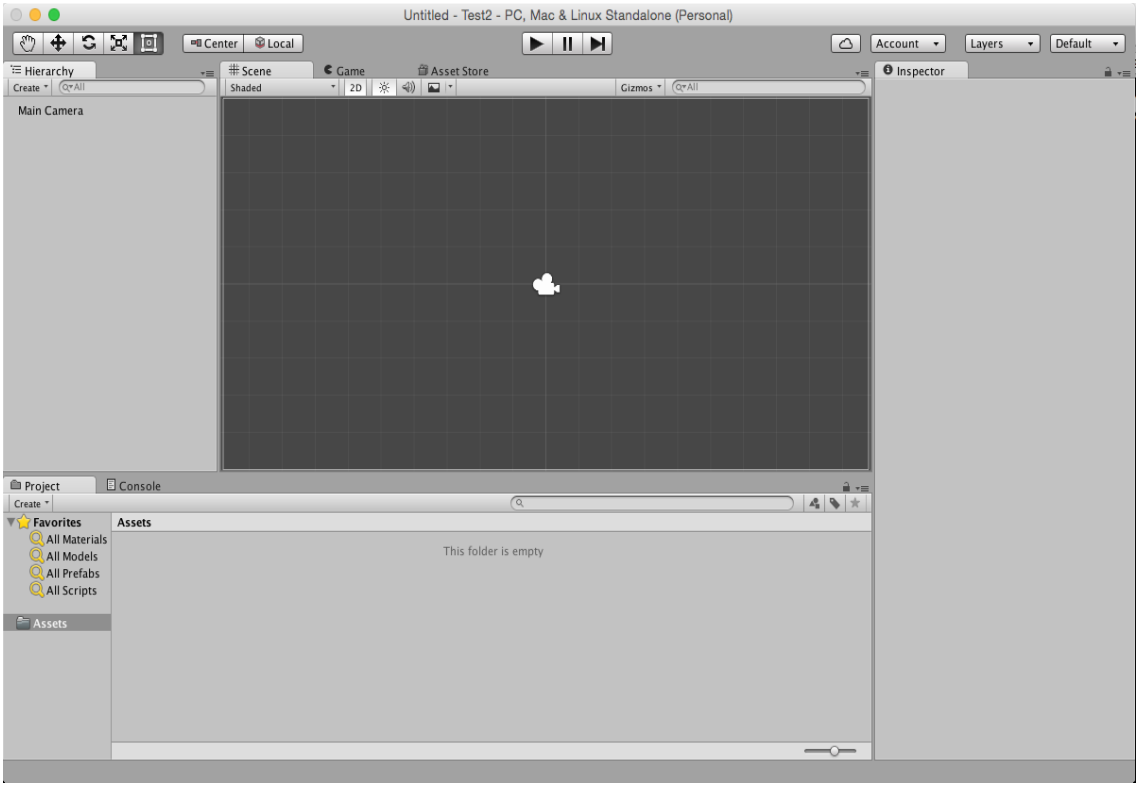
1 Lancez Unity3D (sous Windows ou Linux)

2 Si vous n'avez pas de compte Unity3D, il vous faudra en créer un.

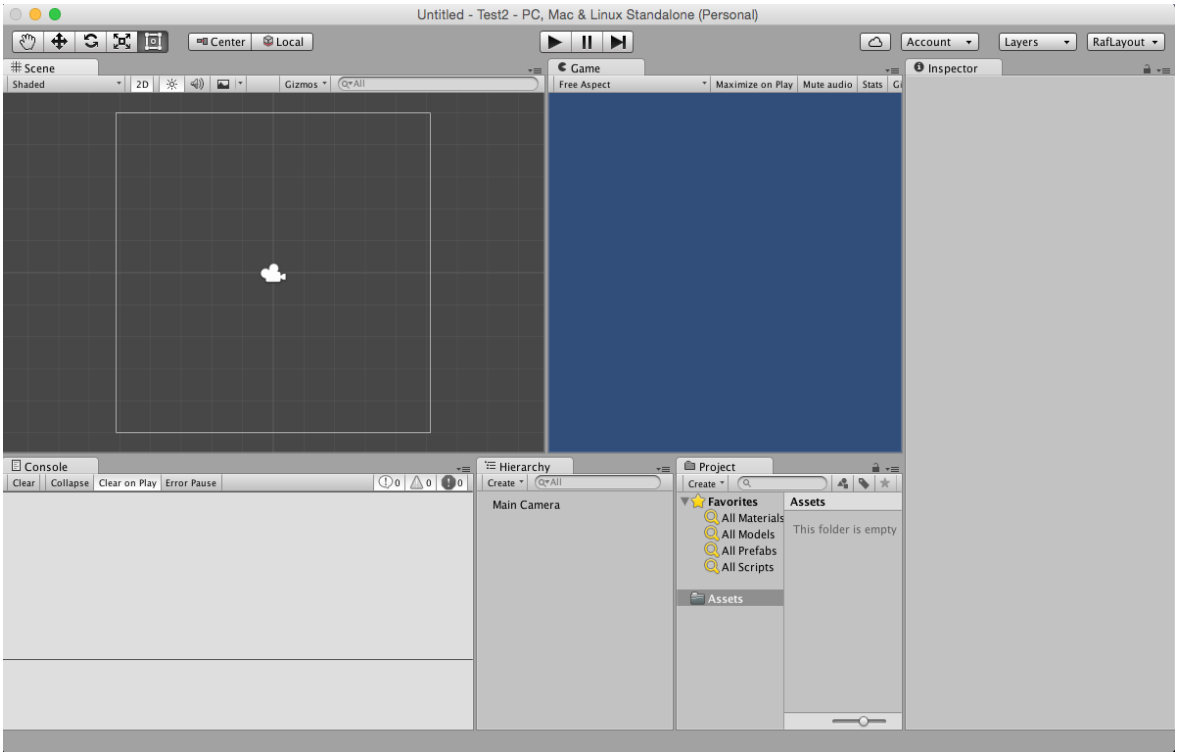
3 Créez un nouveau projet (nom au choix), et choisissez 2D. Vous pourrez faire de la 3D pour votre projet si vous le désirez ; mais dans le cadre de ce TD, nous resterons sur de la 2D afin de nous concentrer sur l'implémentation du son.

Remarque : pour utiliser Unity3D sous Windows dans les salles du CREMI ; si vous vous connectez pour la première fois sous Windows, cela ne devrait pas poser de problème. Sinon, il vous faudra tout d'abord supprimer le répertoire ~/WINDOWS sous Linux avant de rebooter sous Windows. Vous pouvez faire une sauvegarde du contenu de ce répertoire en amont, au cas ou.

4 Lors du premier lancement, la mise en page Unity3D ressemble à cela :



Ce n'est pas très pratique, notamment car *Scene* et *Game* sont par défaut deux onglets exclusifs, alors qu'il est pratique de les avoir en parallèle.
Changez la mise en page (chaque onglet se déplace et s'ancre grâce à du glisser-déposer), pour qu'elle ressemble à la capture suivante



Vous pouvez sauvegarder votre mise en page (layout) pour la retrouver dans vos futurs projets.

5 Vous pouvez maintenant sauvegarder votre première scène (vide).
- 5.1 File → Save Scene
- 5.2 Dans le répertoire « assets », créez un répertoire « _scenes » et enregistrer votre scène dedans. (Le « _ », facultatif, permet de s'assurer que ce répertoire sera toujours en tête lors de l'affichage du contenu de assets)

6 Avant de créer votre premier objet Unity, regardons ce que nous avons déjà.
- 6.1 Dans la hiérarchie, vous pouvez voir *Main Camera*. Toute scène Unity possède au minimum une caméra.
- 6.2 Cliquez sur *Main Camera*, et l'inspecteur va se mettre à jour pour montrer les propriétés de celle-ci
- 6.3 Ce qui nous intéresse pour le moment est la propriété *size* de notre caméra qui est en mode *orthographic* (car en 2D). *Size* représente la moitié de la hauteur, en pixel (la largeur est calculée en fonction de l'aspect du jeu (4:3, 16:9, 16:10, etc.). Ici, une taille de 5 représente une hauteur de 10 pixels. Changez le pour une valeur plus pertinente. Notre personnage faisant 32 pixels de haut, J'ai personnellement mis 80 pour une résolution de 160 de hauteur.
- 6.4 Pour voir à nouveau la zone couverte par la caméra dans l'onglet *scene*, il va vous falloir dézoomer. Vous pouvez également changer l'aspect de votre jeu dans l'onglet *game*, et vous verrez alors la caméra s'adapter dans la *scene*.

  Remarque : il n'y a pas de « main() » dans Unity, mais il y a toujours une caméra principale. Ainsi, si vous avez besoin de gérer des fonctionnalités comme vous le feriez dans un main C/C++, vous pouvez le faire au sein d'un script attaché à cette caméra.

7 Créez votre premier objet Unity

7.1 Dans l'onglet *project*, créez un répertoire « images » sous « Assets ».

7.2 Dans ce nouveau répertoire, faites un clic droit, puis sélectionnez « Import new asset »

7.3 Importez l'image « front.png ». Vous venez d'ajouter votre premier sprite à votre jeu.

Remarque : vous pouvez désormais importer des nouveaux sprite en les copiant-collant directement dans le repertoire « images ».

7.4 En cliquant sur ce sprite, vous pouvez voir ses propriétés dans l'inspecteur. Vous avez notamment la variable *pixels per unit*, fixée par défaut à 100. Cela signifie qu'il faut 100 pixels sur l'écran pour afficher un pixel du sprite. Changez cette valeur à 1 si vous voulez espérer voir votre sprite dans une taille raisonnable ;-)

7.5 Puisque nous travaillons en « pixel art », il faut éviter qu'un filtre visuel ne lisse de trop les contours du personnage. Passez la valeur « filter mode » à « no filter » et la compression à « high quality ».

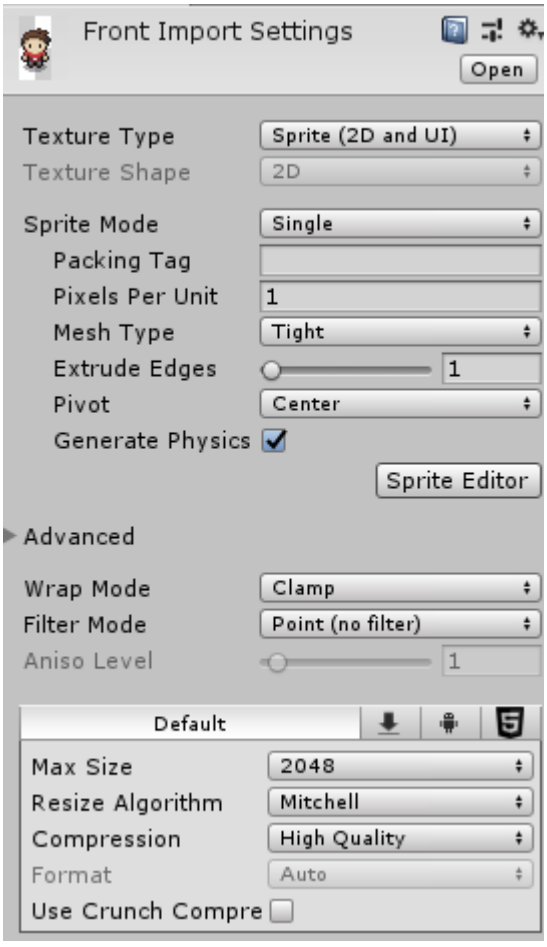
7.6 Puis cliquez sur *apply*

7.7 Glissez votre sprite dans l'onglet *scene*. Vous venez de créer votre premier gameobject, qui apparaît dans la hiérarchie sous le nom « Front ».

7.8 Renommez « Front » en « Player ».

7.9 Cliquez sur l'objet *Player* dans l'onglet *hierarchy*, et vous pouvez jouer avec la couleur, la position, la taille, etc.

7.10 Chaque *gameobject* contient un composant *Transform* qui spécifie la position, la taille et la rotation.



8 Nous allons maintenant déplacer notre player

8.1 Cliquez sur l'objet P*layer* dans l'onglet *hierarchy*

8.2 Dans l'inspecteur, faites « add component »

8.3 Tapez « Move », et sélectionnez « new script » en C#.

8.4 Double-cliquez sur « Move », cela devrait vous ouvrir un éditeur de code source.

Remarque : sous Windows, l'éditeur est soit Visual Studio, soit MonoDevelop. Si vous voulez choisir l'un par rapport à l'autre, allez dans « edit → preferences », et changez la variable « external script editor » sous « external tools »

8.5 Le script *Move* est en partie pré-rempli

8.5.1 La classe *Move* est créée, et dérive de MonoBehaviour. MonoBehaviour est la classe de base dont dépendent tous les objets Unity, notamment pour la création et la mise à jour (à chaque frame).

8.5.2 Les fonctions Start() et Update() sont présentes. Start() est appelée une seule fois, lors de la création de l'objet. Update() est appelée à chaque frame du jeu (généralement entre 30 et 60 fois par seconde)

8.5.3 Remarque : il est souvent plus judicieux d'utiliser la fonction Awake() (à rajouter à la main) plutôt que Start() pour l'initialisation. En effet, un script peut être désactivé dans l'inspecteur (en décochant la case à coté de son nom). Dans ce cas là, la fonction Start() n'est pas appelée tant que le script est désactivé, alors qu'Awake() sera appelée de toute façon.

8.6 Modifiez le update() de la façon suivante :

```
void Update () {
        if (Input.GetKey (KeyCode.LeftArrow)) {
            gameObject.transform.Translate(new Vector3(-100.0f *
            Time.deltaTime, 0, 0));
        }
    }
```

Remarque : Time.deltaTime donne le temps réel passé entre deux appels consécutifs de cette fonction Update()

Remarque : la compilation est automatique sous Unity. Lorsque vous sauvegardez votre fichier, il vous suffit d'aller sur l'éditeur, et de regarder dans la console s'il y a une erreur, un warning, ou rien de tout cela.

8.7 Dans l'éditeur, appuyez sur Play, et testez votre script.

9 Essayons d'être plus générique

9.1 Dans votre classe *Move*, rajoutez la ligne suivante :

```
public class Move : MonoBehaviour {
      public float m_speed = 100.0f;
```

9.2 Modifier le update() en conséquence

```
void Update () {
        if (Input.GetKey (KeyCode.LeftArrow)) {
            gameObject.transform.Translate(new Vector3(-m_speed
            * Time.deltaTime, 0, 0));
        }
    }
```

9.3 Dans Unity, sélectionnez votre objet *Player*. Vous pouvez maintenant accéder à la variable « speed » directement dans l'inspecteur, et tester différentes vitesse sans toucher au code source. Cela est possible car m_speed a été déclarée « public ».

9.4 Écrivez le code pour les quatre directions.

10 Essayons maintenant d'adapter graphiquement le personnage selon la direction donnée. Nous allons voir deux manières de récupérer des informations présentes dans les autres composants : par glisser-déposer, et par accès code source.

10.1 Importez les images « back.png » et « side.png ». Pensez à mettre à jour la variable « pixels per unit »

10.2 Rajoutez dans votre script « Move » les variables publiques suivantes

```
public Sprite m_frontSprite;
public Sprite m_backSprite;
public Sprite m_sideSprite;
```

10.3 Dans l'inspecteur, glissez-déposez chaque sprite (de l'onglet *projet*) vers la variable correspondante sur le *Player*.

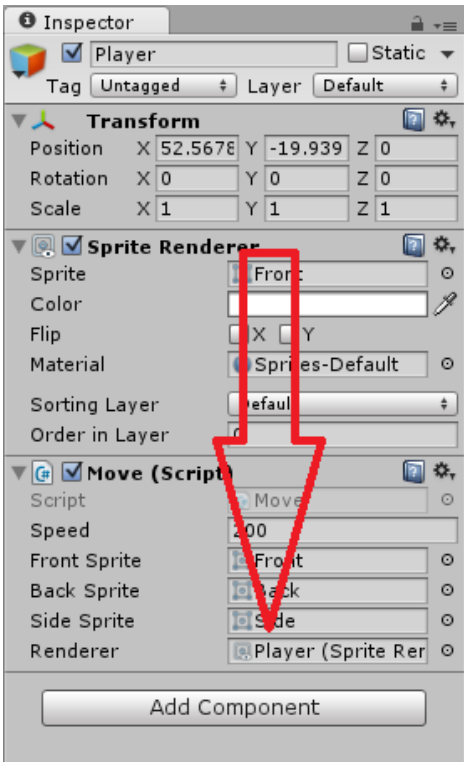
10.4 Maintenant, vous allez devoir mettre à jour le sprite de votre *Player* selon la direction. Il se trouve que vous avez un composant *Sprite Renderer* que vous pouvez utiliser pour changer le sprite visible. Mais il vous faut récupérer les information de ce composant. Voici deux méthodes pour le faire.

10.4.1 La première est de le faire à travers le code source. Pour cela, vous pouvez utiliser la fonction GetComponent<type>( ). *gameObject* retourne le game object sur lequel le script « Move » est appelé.

```
SpriteRenderer currentRenderer = gameObject.GetComponent<Spri
teRenderer> ();
```

10.4.2 La deuxième est de créer une variable publique dans votre script « Move », et d'y glisser-déposer le SpriteRenderer correspondant.

```
public SpriteRenderer m_renderer;
```



La deuxième solution fonctionne également entre game objets, si vous souhaitez qu'un objet récupère les données d'un autre.

10.5 Maintenant, vous pouvez changer votre sprite dans le update() selon la direction prise par le *Player*, en mettant à jour currentRenderer.sprite (ou m_renderer.sprite). Vous aurez sûrement besoin de mettre également à jour la variable flipX.

11 Nous allons terminer en récupérant des pièces, et en affichant un score.

11.1 Importez l'image « coin.png » dans vos assets (et mettez à jour les variables *pixels per unit, filter et compression*) et glissez-déposez ce nouveau sprite dans votre scene.

11.2 Ajoutez aux *Coin* et *Player* un composant *Circle Collider 2D*. Vous verrez alors dans la *scène* un cercle vert, qui correspond à la zone de collision de chaque objet.

11.3 Pour que la collision soit détectée au sein du *Player*, il faut également lui ajouter un *RigidBody2D*. Passez la variable *Gravity Scale* à 0 de votre RigidBody2D pour éviter que votre *Player* ne soit attiré vers le bas de l'écran.

11.4 Créez un nouveau script C# *PlayerXP* pour votre *Player*, et ajoutez-y une variable entière « m_score », publique.

11.5 Implémentez la fonction « *void OnCollisionEnter2D (Collision2D other)* » qui est appelée par Unity lorsque le collider2D de l'objet courant entre en contact avec le collider2D d'un autre objet de la scène. Rajoutez la ligne « Destroy(other.gameobject) », incrémentez votre variable « m_score » et testez.

11.6 Logiquement, cela fonctionne, mais ce n'est pas une solution acceptable en l'état. En effet, tous les objets entrant en collision avec le *Player* seront détruits, sans distinction. Nous allons donc utiliser les mécanismes de *tag* pour résoudre ce problème.

11.7 Cliquez sur l'objet *Coin* dans la hiérarchie, et dans la partie supérieure de l'inspecteur, déroulez le menu « Tag », et faites « Add Tag ». Rajoutez un tag que vous appellerez « coins », et choisissez le dans le menu déroulant. Votre objet a donc maintenant cette dénomination attachée à lui.

11.8 Dans la fonction « OnCOllisionEnter2D », vous pouvez maintenant accéder au tag de « other » par « other.gameobject.tag », et effectuer une comparaison pour être certain que seuls les objets taggés « coins » seront bien détruits.

12 Enfin, nous allons afficher le score grâce à la GUI, et ce sera la fin de cette prise en main d'Unity. A la différence des autres objets de la scène, qui dépendent de la position de la caméra, la GUI d'Unity apparaît en mode « tête haute » à l'écran, ce qui veut dire que ses éléments restent présents au même endroit, peu importe la position de la caméra.

12.1 Dans la hiérarchie, faites un clic droit, et sélectionnez UI → Text. Vous devez voir apparaître l'objet « Canvas » comme parent de « Text ». Tous les éléments de l'UI doivent appartenir à un « Canvas ». Il y a également l'objet « EventSystem » qui s'occupera de gérer les interactions avec la GUI.

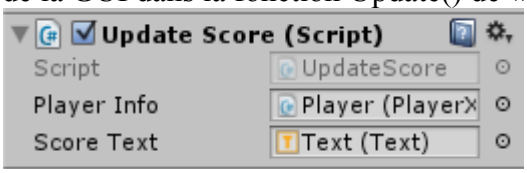
Remarque : comme pour le « Canvas » ici, il vous est possible de spécifier à des objet Unity qu'ils sont *enfants* d'autres objets, en glissant-déposant un objet sur son *parent* dans la hiérarchie. Cela signifie que sa position, rotation et taille de celles de son *parent*. Cela est notamment utile si vous désirez faire un élement de gameplay qui soit constitué de plusieurs sous-ensembles (par exemple, un personnage qui tient une arme, la position de l'arme est relative à la position du personnage ; il est donc normal qu'elle soit « enfant » de « personnage »).

12.2 Déplacez le texte où vous le voulez sur l'écran, dans l'onglet *scene*. Pour bien gérer sa GUI, il est possible de spécifier des pivots et ancres dans l'inspecteur (Rect Transform), mais nous nous couvrirons pas cela dans cette introduction.

12.3 Ajoutez à votre objet texte un nouveau script C#, que vous pouvez appeler « UpdateScore ».

12.4 A l'intérieur de ce script, ajoutez une variable publique de type PlayerXP et une variable publique de type Text (pour cette dernière, vous devrez également inclure UnityEngine.UI).

12.5 Dans l'inspecteur Unity, il ne vous reste plus qu'à glisser-déposer les bons composants dans ces variables nouvellement créées, puis à mettre à jour le texte de la GUI dans la fonction Update() de « UpdateScore ».

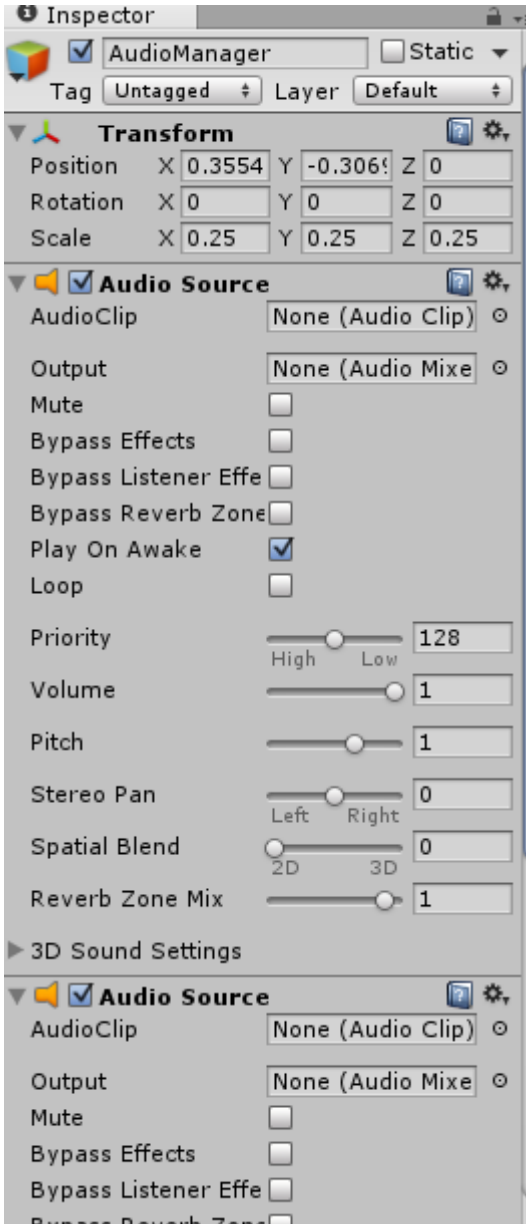


13 Il reste une modification à apporter pour que le code fonctionne de manière pertinente. En effet, déplacer un joueur à partir de son « transform » ne permet pas de bien gérer les collisions avec l'environnement (si l'on désire, par exemple, bloquer un joueur avec un mur). En effet, un déplacement avec « transform » téléporte le joueur à la nouvelle position. Une meilleure solution est en réalité d'utiliser le Rigibody2D (à récupérer comme variable publique, ou avec un GetComponent<RigidBody2D>()), et d'utiliser la fonction MovePosition, ou AddForce de ce RigidBody2D. Ces fonctions doivent être utiliser dans un « FixedUpdate() » au lieu d'un « Update() » (FixedUpdate comportant toutes les fonctions qui agissent sur la physique de l'objet). Dans ce cas, Time.fixedDeltaTime doit être utilisé à la place de Time.deltaTime.

Et voilà ;-)

2 – Création d'un moteur audio (singleton)

1. Créez maintenant un « audio manager »
   1.1 Dans la hiérarchie, faites « create empty » et renommez l'objet créé en « AudioManager ».
   1.2 Sélectionnez « AudioManager », ajoutez deux *audio sources* (une sera pour le son, l'autre pour la musique).



   1.3 Importez des sons
      1.3.1 Dans l'onglet *project*, rajoutez un répertoire « sounds » dans les assets.
      1.3.2 Y importer *coin.wav* et *TownThemeCC0CynicMusic.mp3*.

   1.4 Pour tester rapidement, cliquez sur « AudioManager ». Dans l'inspecteur, cliquez sur le rond pointé à coté de l'un des deux « audio clip », et choisissez « TownThemeCC0CynicMusic ». Lancez votre jeu, et testez les paramètres en direct (pitch, volume, pan, etc.). Ensuite, cliquez à nouveau sur le rond pointé, et supprimez le clip (c'était juste pour tester).

2 Créez votre singleton « AudioManager » utilisable par tous les objets de la scène

2.1 Sur l'objet « AudioManager », faites « add component », et créez un script C# AudioManager. Double-cliquez dessus.
Vous allez faire un singleton static, que vous pourrez alors appeler facilement dans tous les autres scripts C#.

2.2 Remplacez la fonction Start() par les lignes suivantes :

```
public static AudioManager instance = null;

void Awake() {
        if (instance == null) {
                instance = this;
                DontDestroyOnLoad (gameObject);
        }  else {
                Destroy (gameObject);
        }
}
```

2.3 Maintenant, au sein de n'importe quel composant (par exemple “playerXP”) il est possible d'appeler votre manager grâce à AudioManager.instance.

Remarque : DontDestroyOnLoad signifie que votre AudioManager restera présent et intact, même si vous changez de scène. Il serait détruit sinon.

3 Définissez le comportement de votre AudioManager.

3.1 Ajoutez deux variables représentant le flux de la musique et du son

```
public AudioSource m_soundStream;
public AudioSource m_musicStream;
```

3.2 Dans l'inspecteur Unity maintenant, vous avez deux variables pour chaque flux. Vous pouvez leur associer les audiosources de l'objet par glisser-déposer.

3.3 Dans le script AudioManager, ajoutez

```
public void PlaySound(AudioClip soundClipToPlay) {
    m_soundStream.clip = soundClipToPlay;
    m_soundStream.Play ();
}
```

4 Testez

4.1 Dans votre script “PlayerXP” de “Player”, rajoutez une variable AudioClip publique. Associez le son “coin”. Et jouez le son lorsque le joueur récupère une pièce (AudioManager.instance.PlaySound(votreAudioClip)).

5 A vous...

5.1 De créer la fonction PlayMusic, avec un paramètre booléen “loop” indiquant si la musique doit être jouée en boucle ou non.

5.2 De créer une fonction pour modifier le volume du son et de la musique.

5.3 De créer une fonction pour modifier le pitch, jouer la musique à l'envers, etc.

5.4 Autre : soyez créatifs !

6 A vous également...

6.1 De regarder le fonctionnement de *Instantiate* et de créer un coinManager qui fait apparaître régulièrement des pièces sur l'écran (à vous de décider comment)

6.2 De créer un deuxième personnage qui, lui, se déplacera tout seul (à vous de décider comment)

6.3 Autre : soyez créatifs ! ;-)