UNITY

Je crée mon jeu vidéo

14 ans et plus

Carnet de l'enseignant·e

CodeNPlay

Financé par l'Union européenne
NextGenerationEU

DigiLab

UNITY, C# ET VISUAL STUDIO

Unity
Unity est un moteur de jeu, c'est à dire un ensemble de composants logiciels qui servent à simuler l'univers du jeu avec ses mécaniques, interactions, visuels et sons.

Les moteurs de jeu permettent à des créateurs·trices de fabriquer des jeux sans avoir à programmer l'ensemble des systèmes nécessaires. Ces derniers étant pris en charges par le moteur.

Unity est un moteur de jeu utilisé dans l'industrie du jeu vidéo mais aussi dans l'animation 3D. Il est l'un des deux mastodontes du marché avec le moteur Unreal. Il profite d'une communauté très active et dispose donc de beaucoup de tutoriaux et ressources disponibles en ligne.

C#
C# est un langage de programmation orienté objet développé par Microsoft. Il est ici utilisé pour créer des scripts altérant le comportement des éléments qui seront présents dans le jeu.

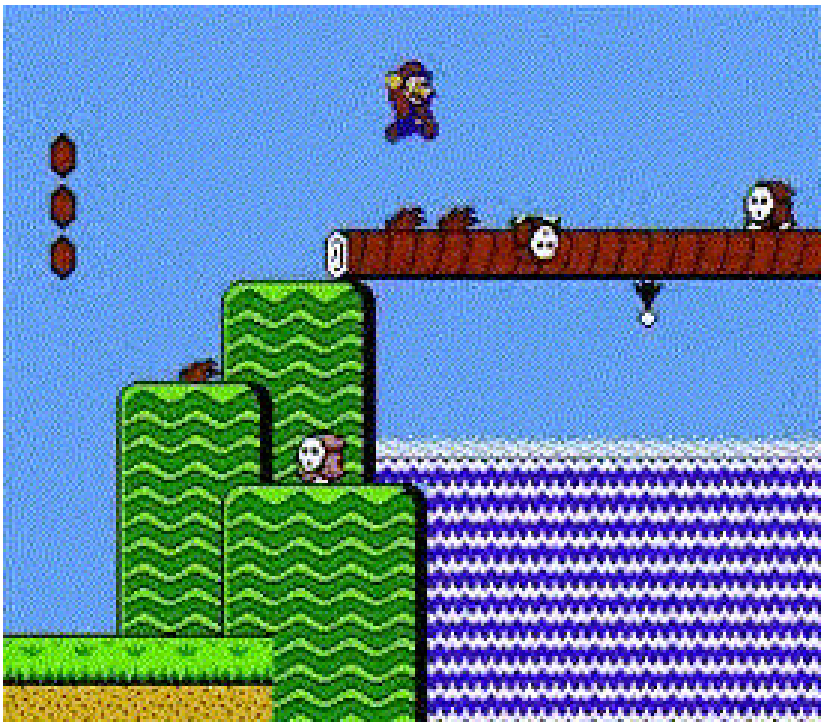
Visual Studio
Les programmeurs·euses utilisent généralement un IDE, un environnement de développement intégré, c'est à dire une programme dédié à la programmation. Il permet d'écrire des programmes, les faire fonctionner, identifie les erreurs de syntaxes et offre bon nombre d'outils supplémentaires.

Dans ce modules nous utiliserons Visual Studio, une suite de logiciels comprenant un IDE pour C#. Il n'est pas nécessaire d'utiliser un IDE pour faire de programmation mais cela offre beaucoup d'avantages et simplifie énormément le travail. Visual Studio a l'avantage de disposer de beaucoup d'outils utiles et d'être gratuit pour sa version communauté.

Objectifs de ce module d'activités
L'objectif de cette suite d'activités est de donner les bases de la création de jeu de plateforme (à la celeste) avec Unity et de permettre à vous et à vos apprenant·es de créer des jeux vidéos 2D et 3D en autonomie.

2D VS 3D

Les premiers jeux vidéo étaient en 2 dimensions, c'est-à-dire que les éléments qui composent le jeu ne pouvaient se déplacer que sur 2 axes. Ceci était dû aux limitations techniques des ordinateurs et consoles de l'époque.



Par la suite on a été capable de créer des jeux en trois dimensions avec des éléments qui se déplacent sur trois axes.

Unity est un moteur de jeu 3D qui gère donc les déplacements sur 3 axes (x, y et z). Ceci est le cas même quand vous créez un projet d'un jeu 2D. Tous vos objets auront leurs positions dans l'espace représentées par 3 valeurs x, y et z.

Les seules différences sont que le projet sera préconfiguré pour rendre plus simple certaines choses spécifiques à la 2D ou la 3D.

CONCEPTS DE PROGRAMMATION

Algorithmes
Un algorithme est une série d'étapes qui résolvent un problème ou accomplissent une tâche. Par exemple, pour faire une tartine, vous pouvez suivre un algorithme simple : prendre une tranche de pain, la toaster, ajouter du beurre et de la confiture. Dans la programmation, les algorithmes sont utilisés pour résoudre des problèmes et effectuer des tâches.

Boucles
Une boucle est une instruction qui répète une action plusieurs fois. Par exemple, pour compter jusqu'à 10, vous pouvez utiliser une boucle qui commence à 1 et se termine à 10. Dans la programmation, les boucles sont utilisées pour exécuter des instructions répétitives.

Conditions
Une condition est une instruction qui vérifie si une condition est vraie ou fausse. Par exemple, si vous voulez savoir si un nombre est pair, vous pouvez utiliser une condition qui vérifie si le nombre est divisible par 2. Dans la programmation, les conditions sont utilisées pour exécuter des instructions différentes en fonction de la situation.

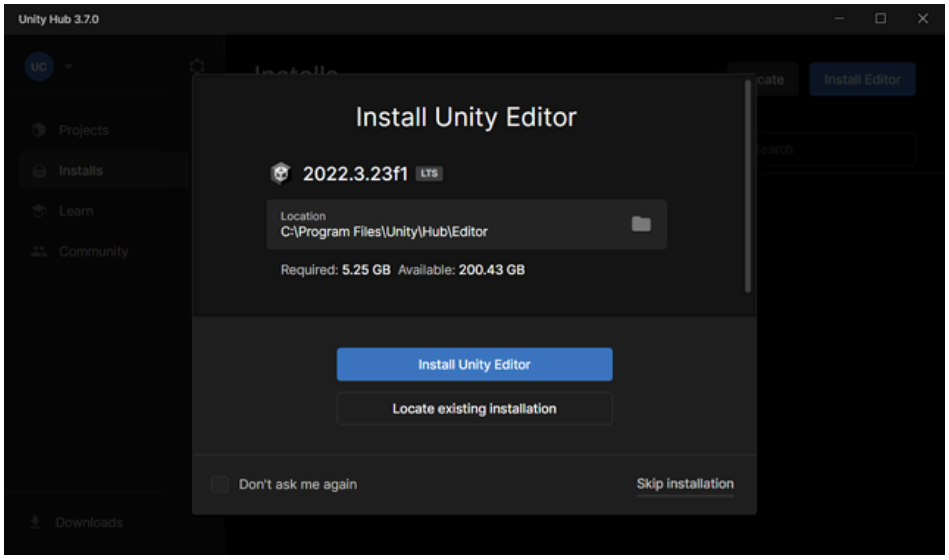
Variables
Une variable est un espace de stockage qui peut contenir une valeur. Par exemple, si vous voulez stocker votre âge, vous pouvez utiliser une variable appelée "âge" et y stocker la valeur 8. Dans la programmation, les variables sont utilisées pour stocker des données et les utiliser plus tard dans le programme.

Fonctions
Une fonction est un ensemble d'instructions qui résolvent un problème ou accomplissent une tâche. Par exemple, pour dessiner un carré, vous pouvez créer une fonction qui dessine un côté, puis appeler cette fonction quatre fois pour dessiner les quatre côtés du carré. Dans la programmation, les fonctions sont utilisées pour organiser le code et le rendre plus facile à comprendre et à gérer.

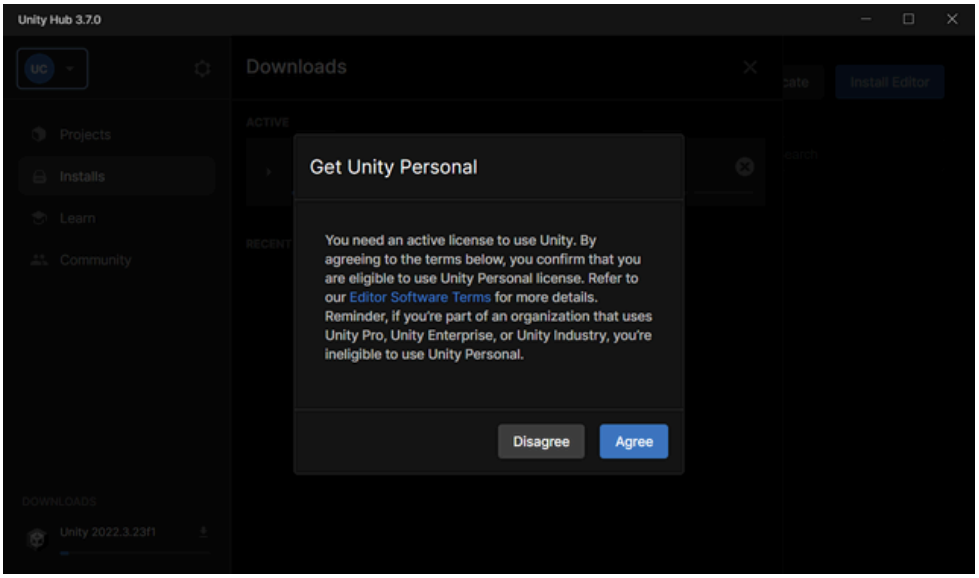
INSTALLER UNITY

Allez sur cette page web et cliquez sur “Téléchargement pour Windows”.

Ouvrez le fichier téléchargé.
Une fenêtre s’ouvrira, autorisez les modifications.
Cliquez sur “J’accepte”.
Cliquez sur “Installer”.
Attendez que l’installation se fasse.
Cochez “Lancer Unity Hub” puis cliquer sur “Fermer”.
Une fenêtre va s’ouvrir, cliquez sur “Create an account”.
Créez votre compte.
Vous recevrez un mail de confirmation, ouvrez le et confirmez votre adresse mail.
Revenez sur la fenêtre Unity Hub et cliquez sur “Sign in”.
Une fenêtre de navigateur s’ouvrira vous demandant de vous connecter à votre compte Unity, faites le.
Un pop-up vous demandant l’autorisation d’ouvrir Unity Hub va s’afficher, acceptez.
La fenêtre Unity Hub va s’afficher.
Cliquez sur “Install Unity Editor”.



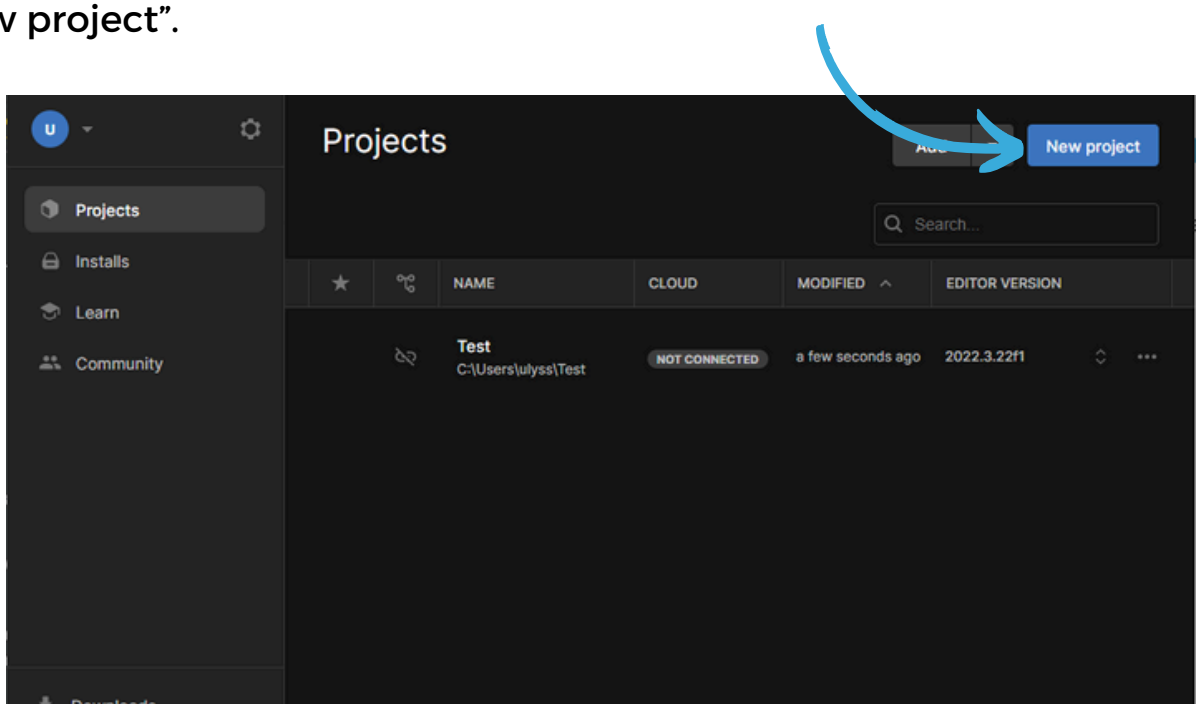
Cliquez sur “Agree”.



Attendez que l’installation se fasse.
Une fenêtre s’ouvrira, autorisez les modifications.
Attendez que l’installation se termine.

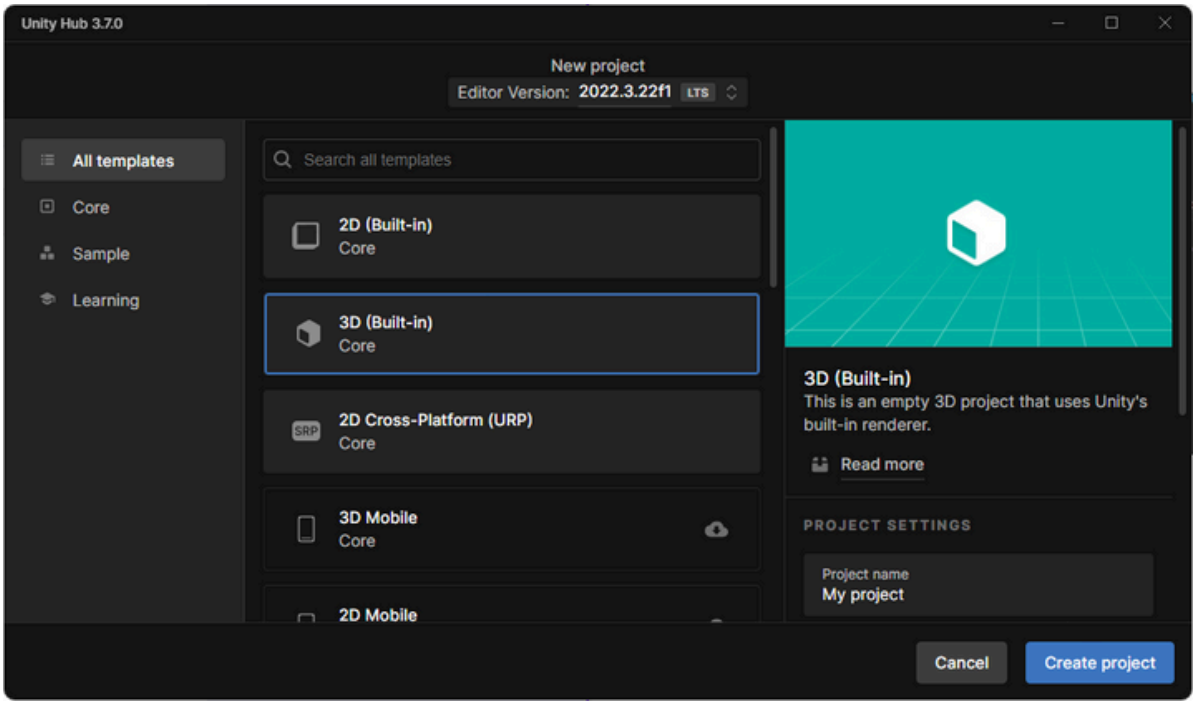
CRÉER UN PROJET

Pour créer un nouveau projet lancez le programme “Unity Hub”, et cliquez sur “New project”.



Sélectionnez le template qui convient à votre projet (pour le début de ce module d’activités, sélectionnez “2D (Built-in)”, cela pré configurera votre projet avec des paramètres en fonction de votre choix.

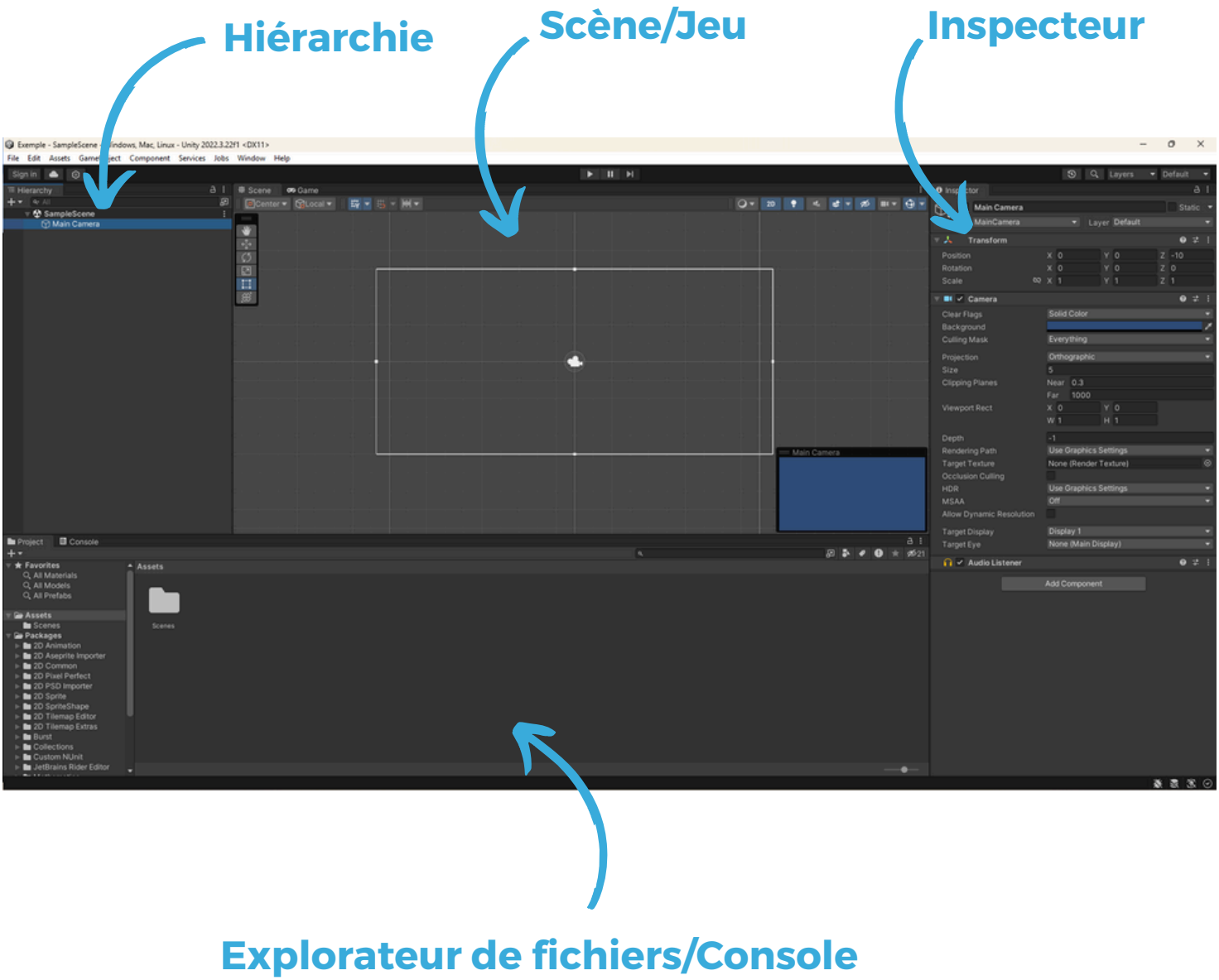
Vous pourrez toujours éditer ces paramètres à votre convenance mais la pré configuration vous fera gagner du temps. Entrez ensuite un nom pour votre projet et cliquez sur “Create project”. Une fenêtre s’ouvrira, attendez que le projet se configure et l’éditeur Unity s’ouvrira sur votre projet.



DÉCOUVERTE DE UNITY

L'éditeur Unity

L'éditeur Unity est un programme qui va nous permettre d'éditer un projet Unity. Il va comprendre plein d'outils utiles à la création d'un jeu.
Ce dernier va être découpé en sous fenêtres ayant chacune leur utilité.



La hiérarchie
Cette fenêtre va afficher tous les éléments qui sont actuellement présents-dans le jeu. Des élément peuvent être dans les fichiers du jeu mais pas encore présent. On ne les retrouvera pas ici. A l'ouverture d'un projet la hiérarchie ne contient que la "Main camera".

Scene/Game
Cette fenêtre contient deux onglets, le premier (Scene) va servir à voir librement les éléments présents dans le jeu, les sélectionner, les faire bouger, etc. L'autre (Game) va afficher ce qui sera vu par le/la joueur/joueuse une fois le jeu lancé.

L'inspecteur est la fenêtre qui va permettre de voir et changer les propriétés d'un élément qu'on aura sélectionné. Cette dernière va afficher des choses très différentes en fonction de l'élément.

L'explorateur de fichiers est la fenêtre qui va permettre de voir et manipuler tous les fichiers utilisés dans le projet.

La console est la fenêtre qui va afficher des messages envoyés par les programmes du jeu lors de son exécution.

TROUVER DES RESSOURCES

Pour créer un jeu vous aurez besoin de nombreuses ressources (images, musiques, bruitages, modèles 3D, etc).

Il existe de nombreux sites qui partagent des ressources payantes ou gratuites. Vous trouverez ici une liste de sites qui partagent entre autres des ressources gratuites.

Assets 3D :

- Itch.io (Gratuits et payants)
- Open Game Art (Gratuits)
- CraftPix (Gratuits et payants)
- GameDev Market (Gratuits et payants)
- GameArt2D (Gratuits et payants)
- Kenney (Gratuits et payants)

Effets sonores :

- Itch.io (Gratuits et payants)
- Open Game Art (Gratuits)
- GameDev Market (Gratuits et payants)
- Kenney (Gratuits et payants)

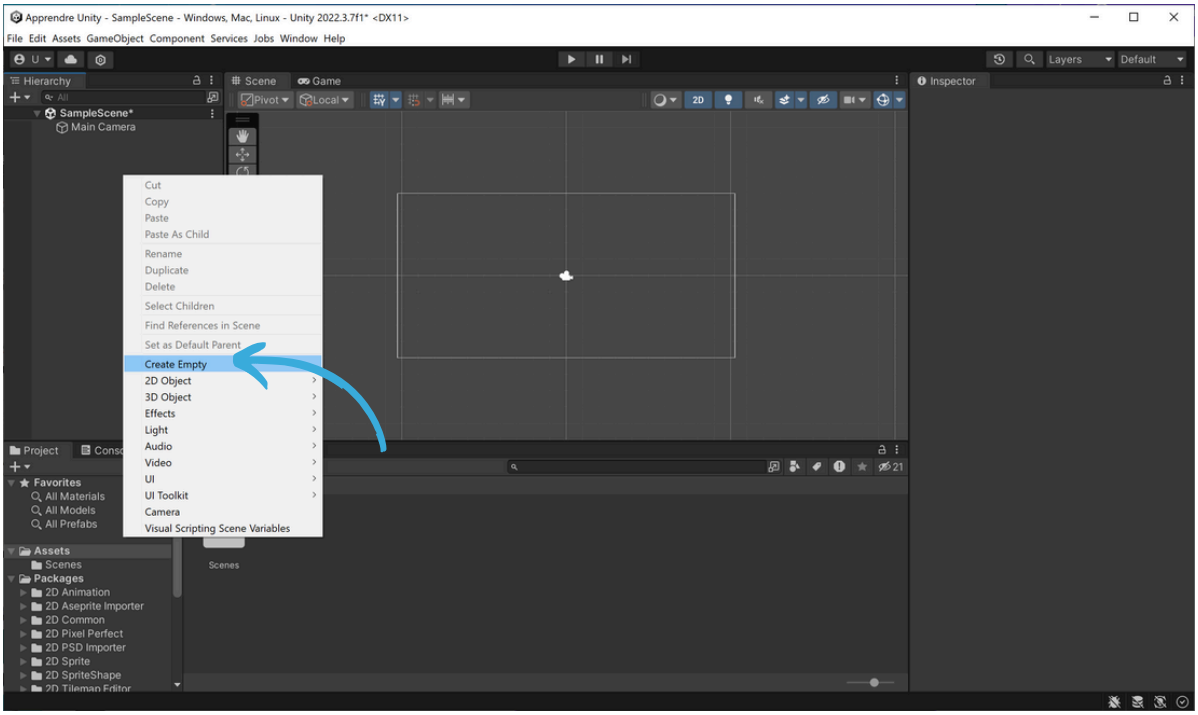
Eléments d'UI :

- Itch.io (Gratuits et payants)
- CraftPix (Gratuits et payants)
- GameDev Market (Gratuits et payants)
- GameArt2D (Gratuits et payants)
- Kenney (Gratuits et payants)

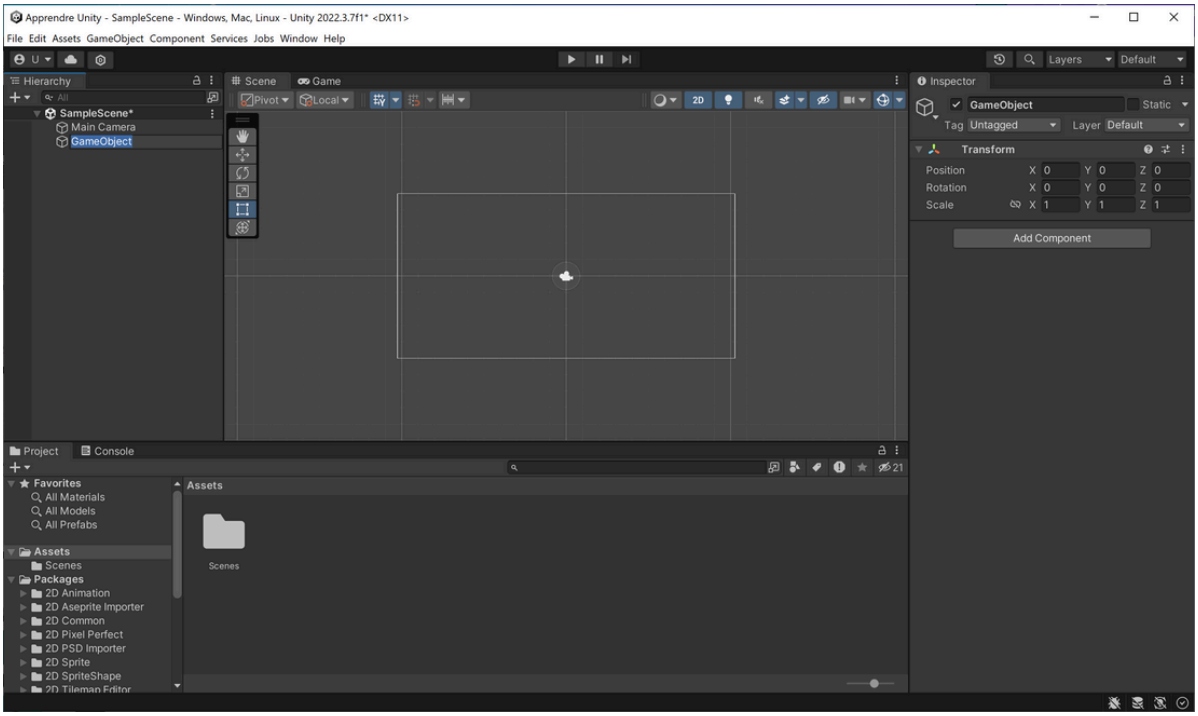
LES GAMEOBJECTS

Game object

La première chose importante à comprendre avec Unity est que tous les éléments que nous allons ajouter à notre jeu sont des Game Objects (littéralement “Objets de jeu”). Pour créer un nouveau Game Object faites un clique droit dans la hiérarchie et cliquez sur “Create empty”.



Ceci va créer un Game Object vide avec son nom surligné en bleu, cela signifie que vous pouvez écrire un nouveau nom (par défaut “GameObject”). Si plus tard vous voulez changer le nom, cliquez deux fois à 1-2 secondes d’intervalle sur l’élément à changer dans la hiérarchie.



LES COMPOSANTS

Les composants
Si vous cliquez sur un GameObject vous pourrez voir dans l'inspecteur ses composants et propriétés. Les composants sont des éléments qui vont donner des propriétés à un GameObject. De base ils comportent tous un composant nommé "Transform" mais il est possible de leur rajouter plein de composants différents en fonction des besoins.

C'est l'une des grandes force du moteur Unity, il permet de rajouter énormément de propriétés préconstruites à nos GameObject et simplifie donc énormément la création.

Le composant Transform
Ce dernier définit pour chaque objet sa position dans l'espace, sa rotation et son échelle à l'aide de différentes valeurs.

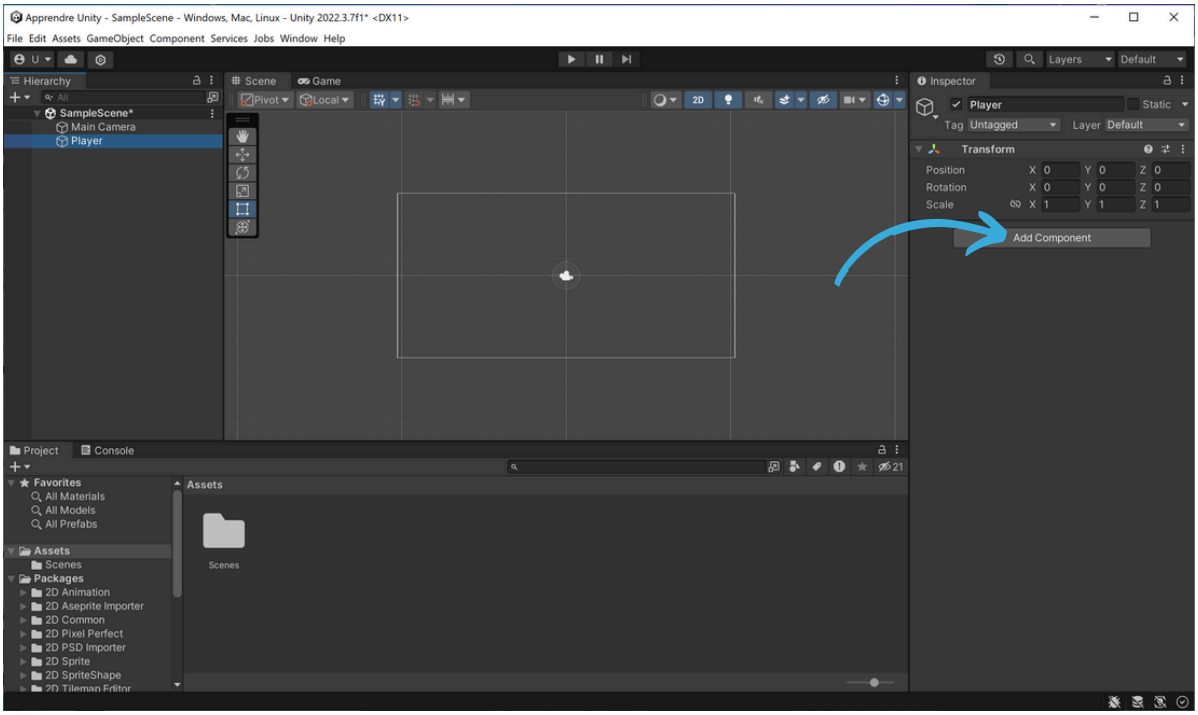
Sa position est définie en utilisant des coordonnées cartésiennes x, y et z. En 2D x correspond à la position sur l'axe horizontal, y sur l'axe vertical et z à l'axe perpendiculaire au plan. Dans les Scripts (voir "Les scripts") les coordonnées sont regroupées dans un Vector3.

Sa rotation est définie en utilisant trois angles qui déterminent dans quelle direction il est orienté, je vous invite a essayer de jouer plus tard avec ces valeurs sur un GameObject avec un Sprite (nous verrons ce dont il s'agit).

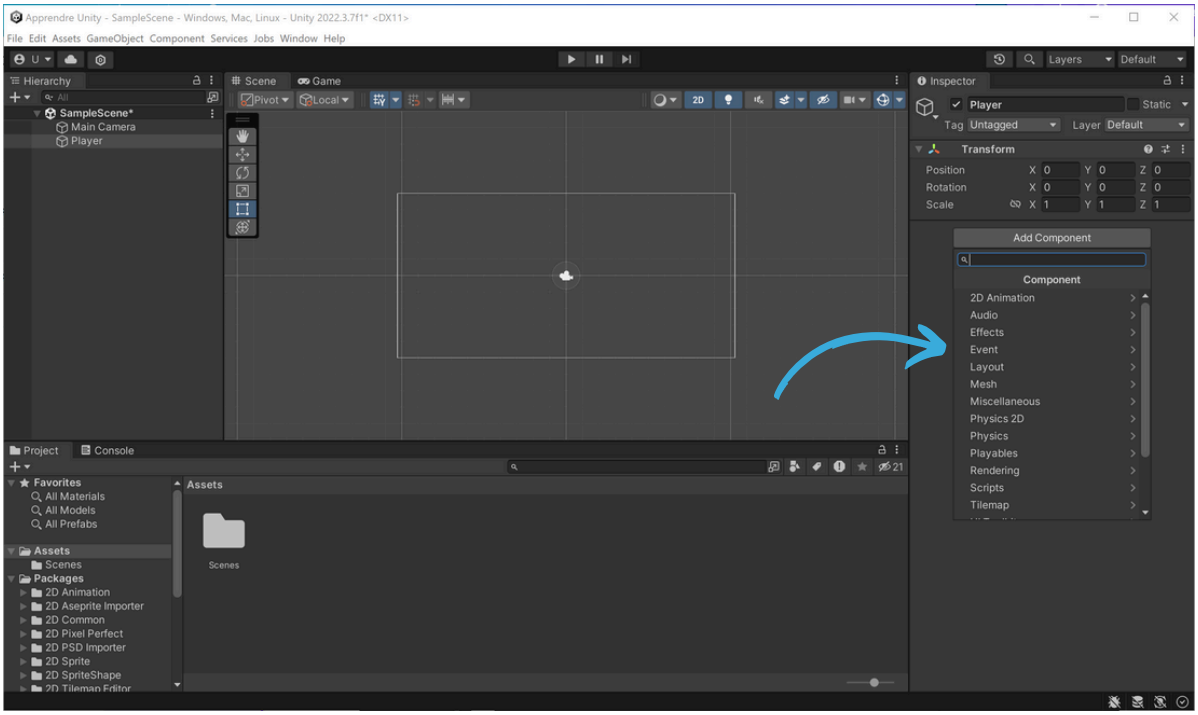
Son échelle (en anglais "scale") correspond à la taille du GameObject, pour un objet vide elle n'a aucune importance mais aura un impact sur certains composants.

Rajouter des composants

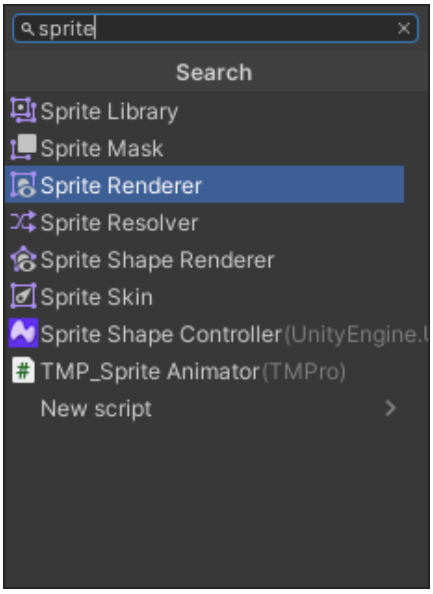
En ayant sélectionné un GameObject vous verrez dans l'inspecteur un bouton "Add Component" celui-ci permet, comme son nom l'indique, de rajouter des composants à votre GameObject.



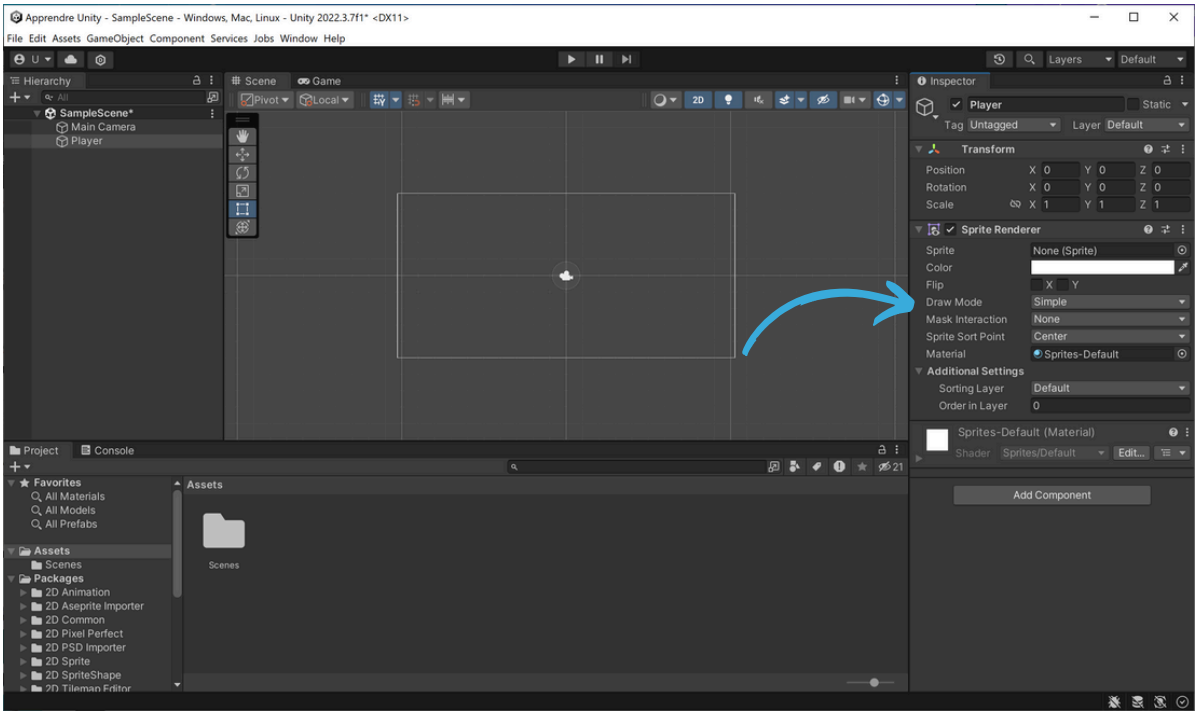
Si vous cliquez dessus vous verrez une fenêtre s'ouvrir avec un menu déroulant et une barre de recherche, elle permet de sélectionner le composant à ajouter.



Pour le moment tapez “sprite”, vous verrez des résultats filtrés en fonction de cette requête. Cliquez sur “Sprite Renderer”.



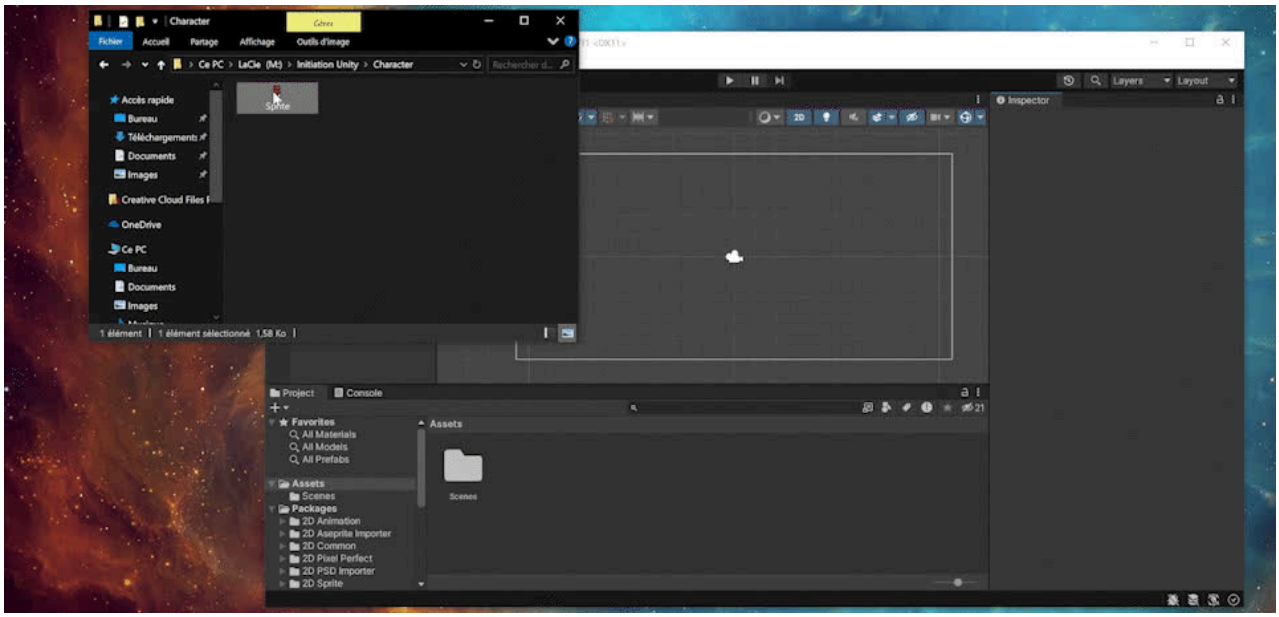
Vous verrez alors que le Sprite Renderer a bien été rajouté à la liste des composants de votre GameObject.



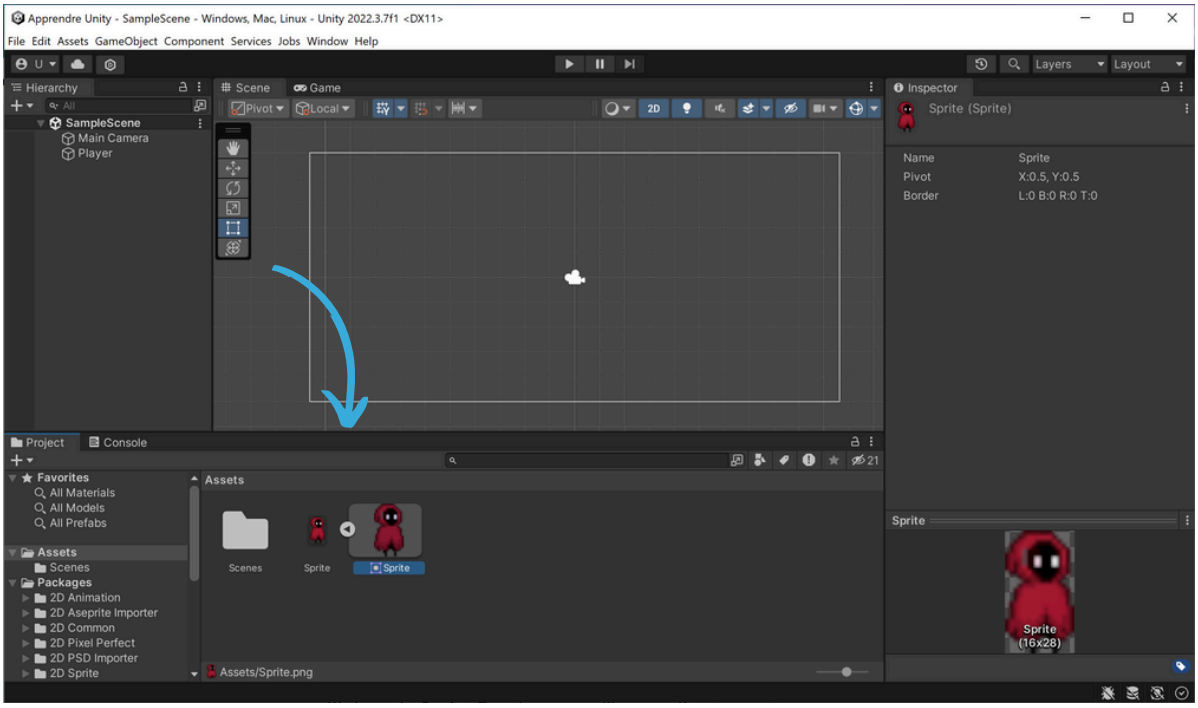
SPRITE RENDERER

Le composant Sprite Renderer (en français : “Moteur de rendu de sprite”) va permettre de donner une apparence à notre GameObject, sous la forme d’une image (sprite) que l’on va définir.
Le nom “sprite” qui en anglais veut dire “lutin” vient des débuts de l’informatique. Du fait que les sprites sont des images qui “flottent” devant un fond on leur a donné le nom des créatures mythologiques.

Pour ce faire, prenez une image que vous aurez téléchargé et glissez-déposez là dans l’explorateur de fichier de Unity, ceci importera l’image dans votre projet.

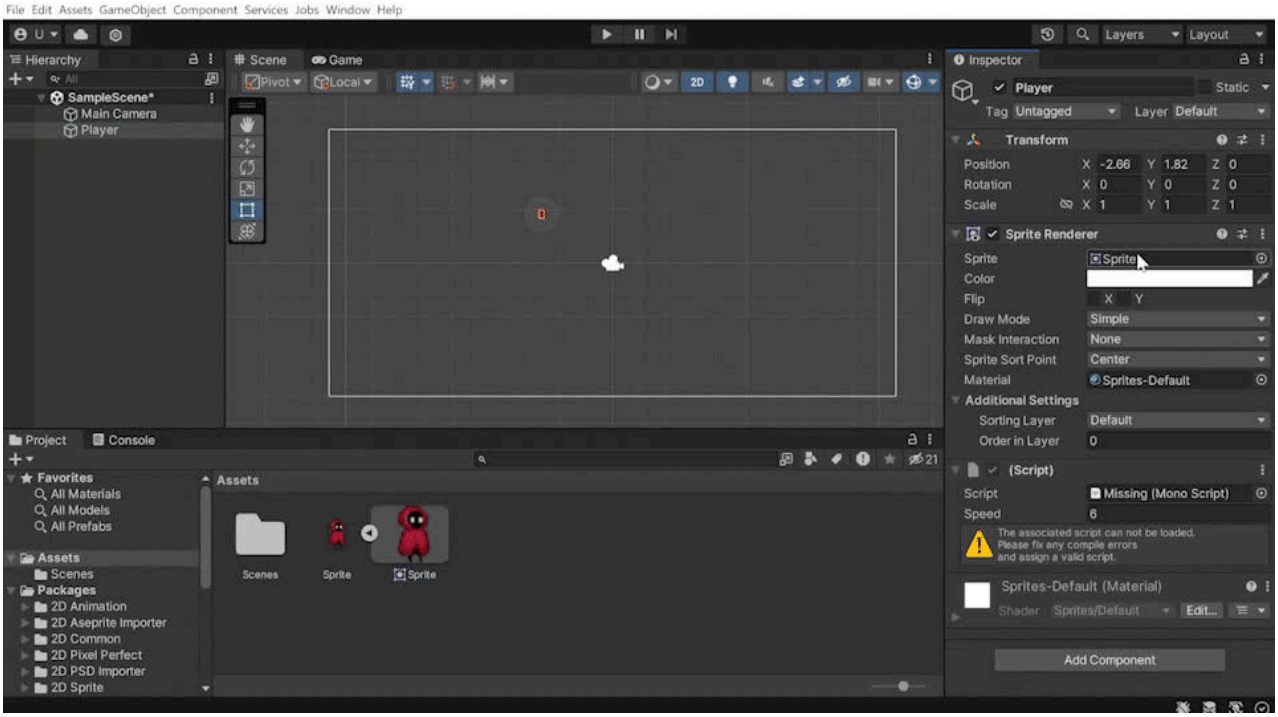


Vous verrez que l’image apparaitra dans l’explorateur de fichier avec une petite flèche à sa droite. Si vous cliquez sur la flèche vous verrez une case s’étendre à côté de votre image elle affichera le Sprite qui aura été créé.



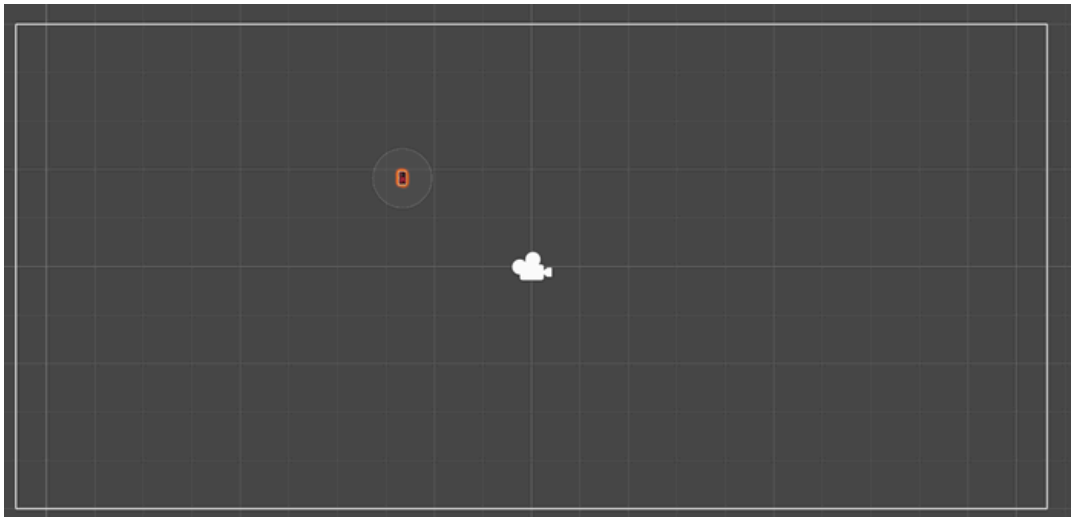
Ce qu'il faut comprendre ici c'est que lorsque vous avez importé votre image, Unity a immédiatement créé un objet de type Sprite. Et c'est le Sprite qui sera utilisé par le Sprite Renderer, pas l'image directement.

Nous allons maintenant utiliser le Sprite pour le faire afficher par le Sprite Renderer. Pour ce faire, sélectionnez votre GameObject, vous verrez dans son composant Sprite Renderer une case "Sprite", glissez-déposez le Sprite vers cette dernière.

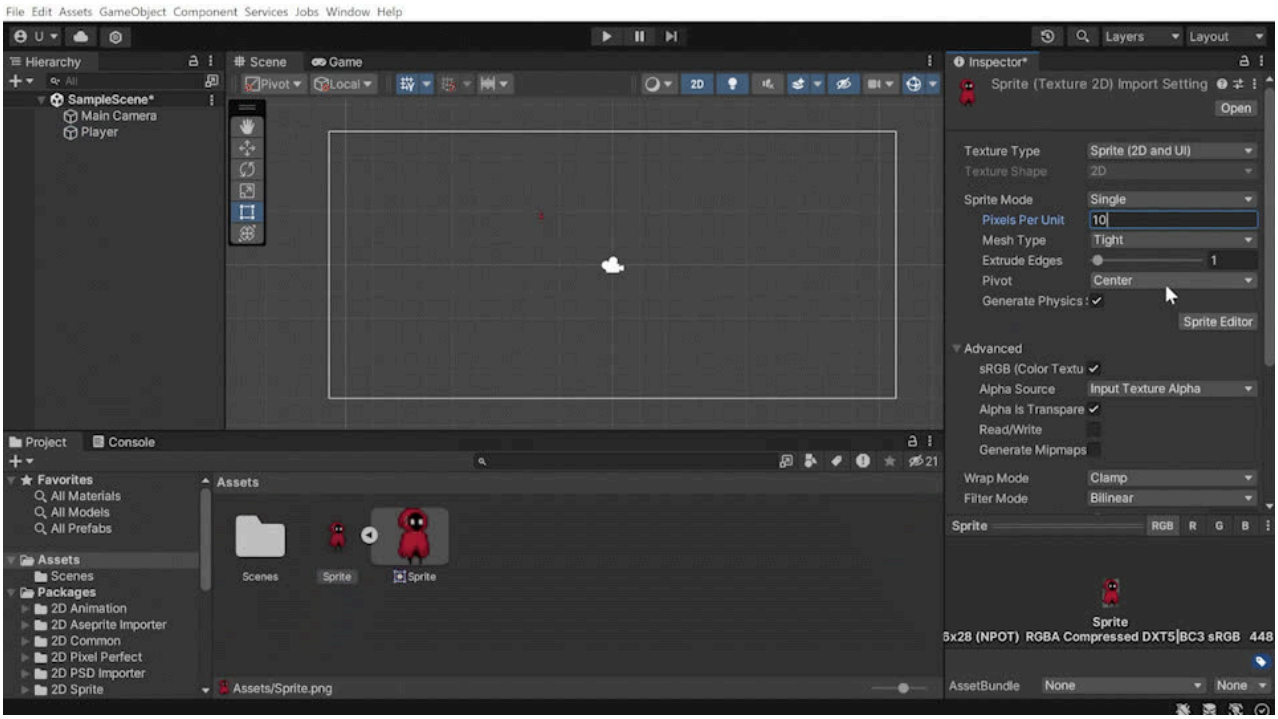


PROBLÈMES COURANTS AVEC LES SPRITE RENDERERS

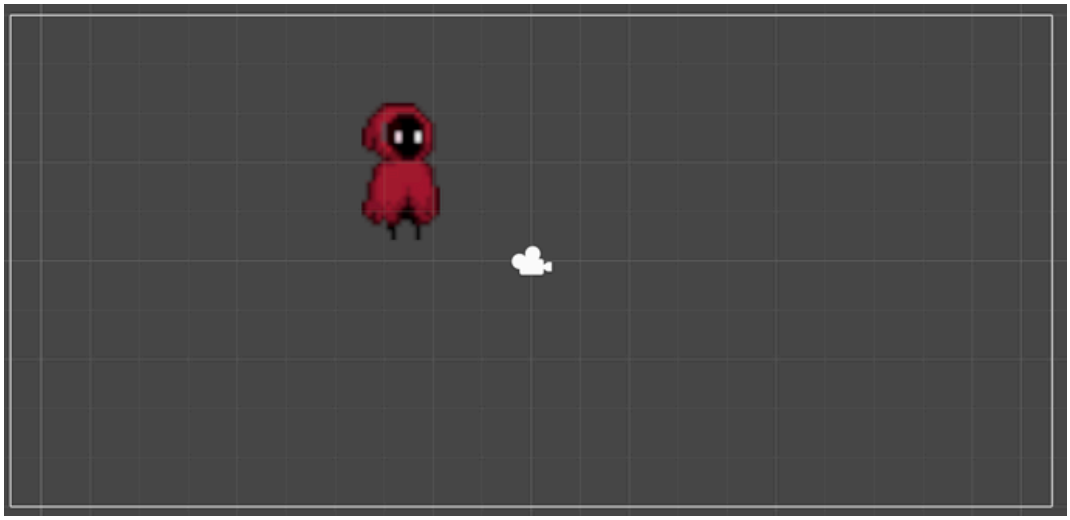
Il y a une série de problèmes courants avec les Sprites qu'il faut être capable de résoudre. Premièrement il peut arriver que le Sprite, une fois affiché, ne soit pas au bonne dimensions. Pour une image avec un faible définition par exemple, le Sprite sera tout petit.



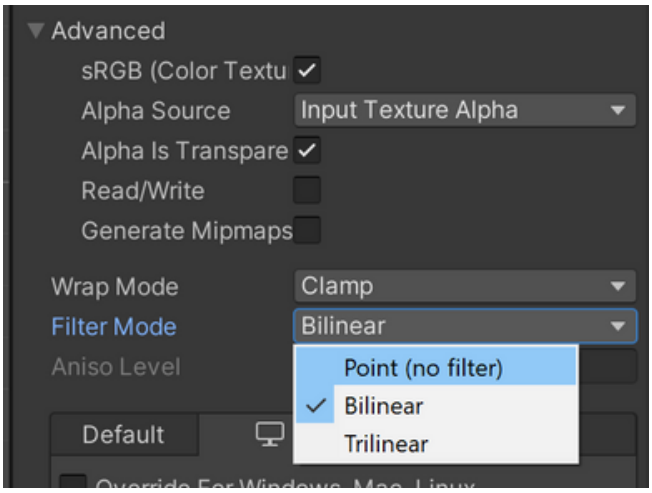
Pour régler ce problème il vous suffit de sélectionner l'image qui a donné le Sprite et d'aller dans l'inspecteur modifier la valeur "Pixels Per Unit" (n'oubliez pas de cliquer sur "Apply"). Cette dernière fait correspondre à un nombre de pixels de l'image une unité de distance dans le jeu. Plus cette valeur sera petite plus le Sprite apparaitra grand.



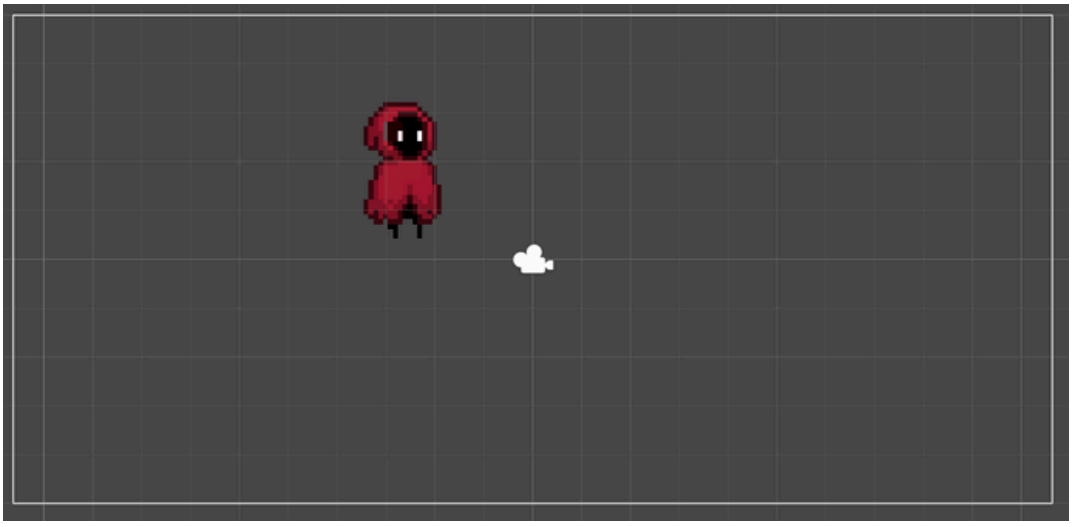
Mettez donc une valeur plus petite puis cliquez sur "Apply" plus bas dans l'inspecteur pour appliquer ces changements.



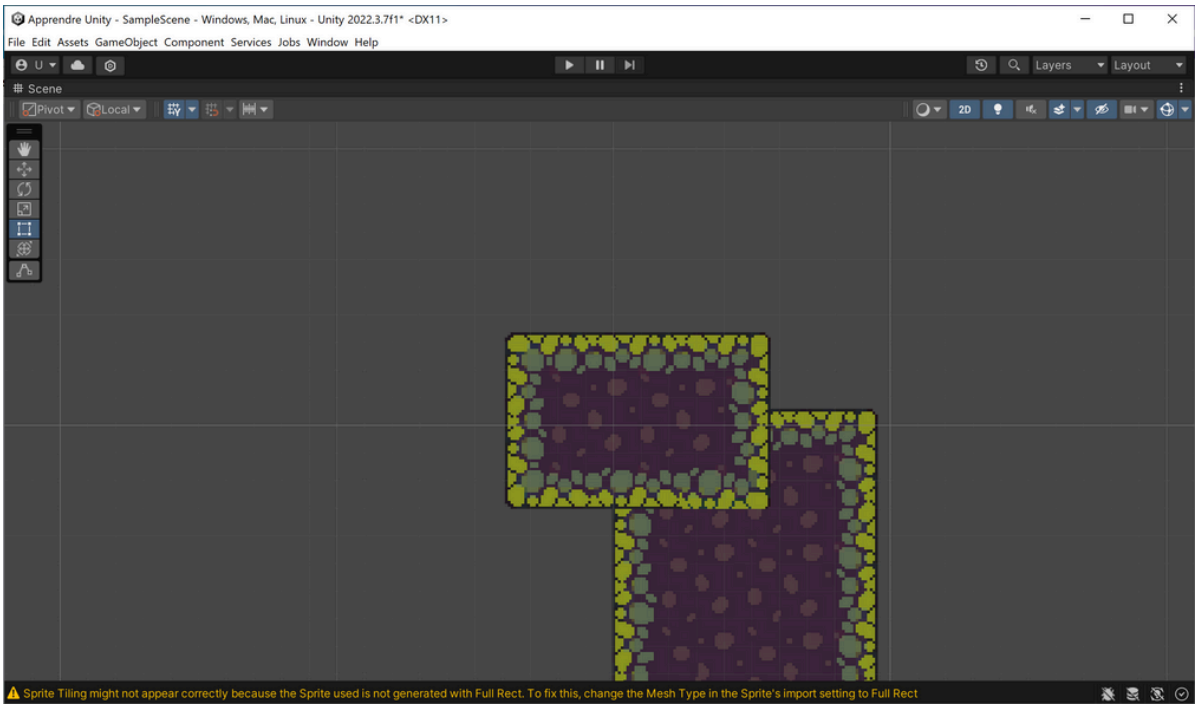
Nous voyons maintenant que les Sprite est à la bonne taille mais il est flou. Pour résoudre ce problème il nous faut changer le paramètre “Filter Mode”. Ce dernier défini comment l’apparence du Sprite (ce qu’on appel sa texture) sera généré à partir de l’image.



Avec le mode Bilinear on va voir un flou sur les images à faible résolution. Pour avoir une texture fidèle à l’image d’origine il nous faut choisir le mode “Point (No Filter)”, n’oubliez pas de cliquer sur “Apply” plus bas.



Si vous superposez plusieurs Sprites vous verrez qu'ils sont affichés comme si ils étaient les uns au-dessus des autres.

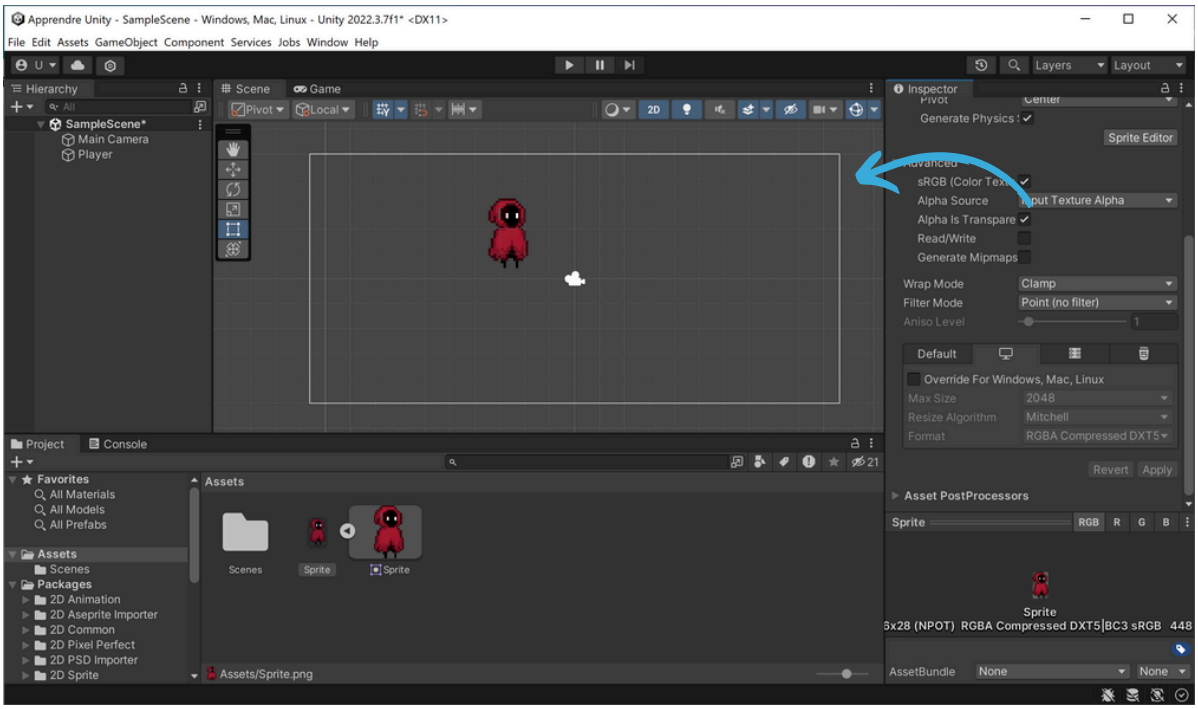


Si les Sprites que vous avez créé sont affichés derrières les autres alors qu'ils doivent être devant ou l'inverse, il vous faudra utiliser les Sorting Layers. Vous trouverez comment faire dans la section "Layers" des fiches techniques

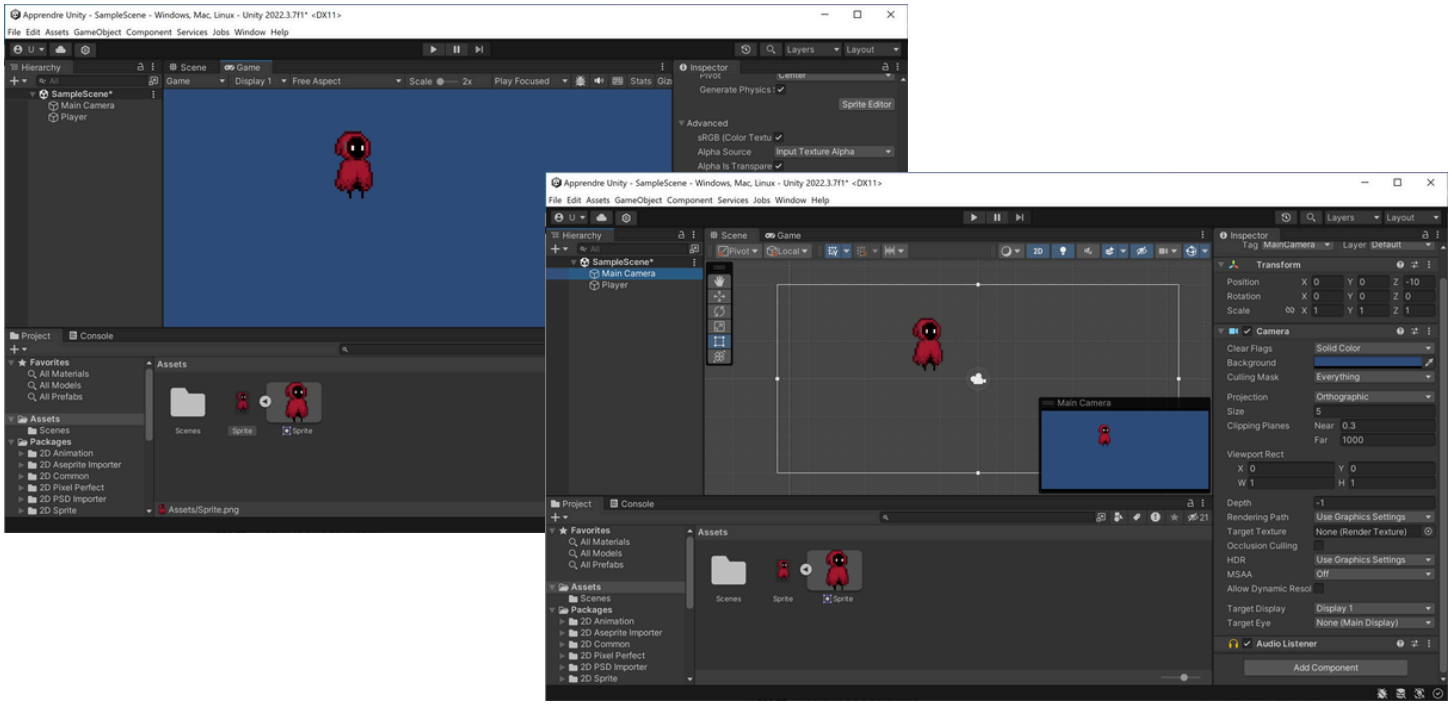
LA CAMÉRA

Parmi tous les GameObject d'un jeu il en est un d'une grande importance, la caméra. C'est ce dernier qui va déterminer ce qui sera visible à l'écran et à quelle position les choses apparaitront.

Si vous regardez votre Scene vous verrez un cadre en fins traits blancs avec en son centre un logo de caméra.

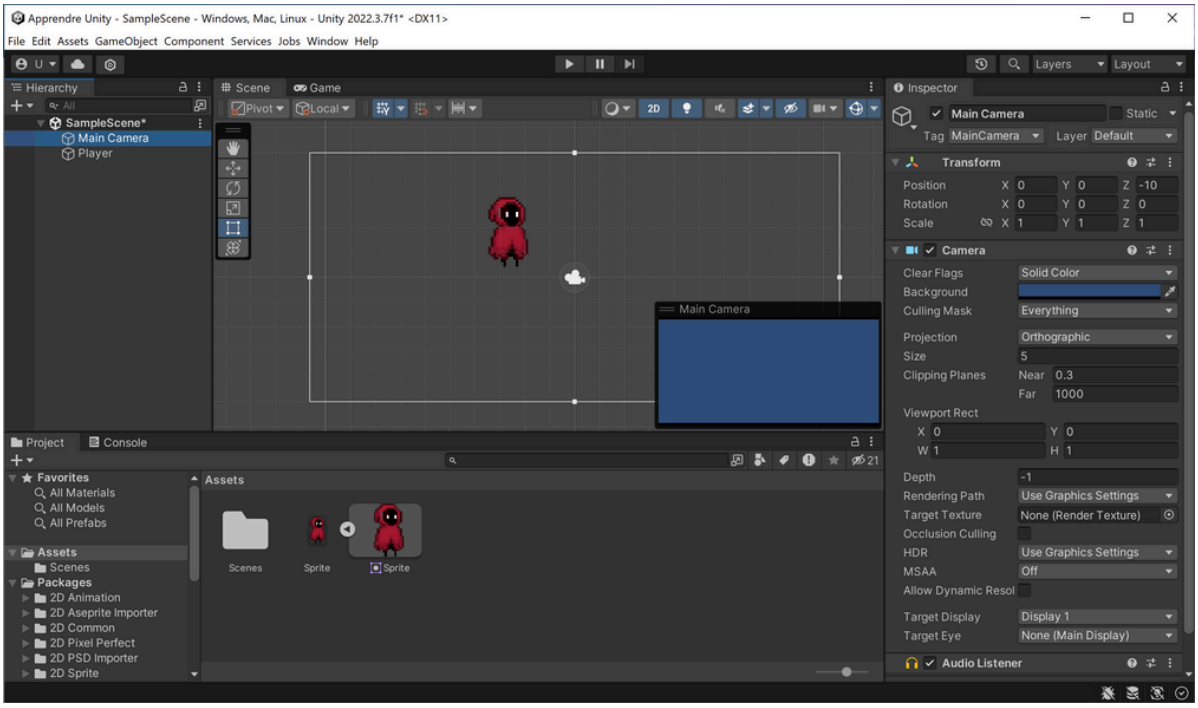


Le cadre blanc encadre tout ce qui sera visible à l'écran du jeu. Pour voir directement ce que voit la caméra, cliquez sur l'onglet "Game" ou sélectionnez la caméra et un petit encadré apparaitra, affichant ce que voit la caméra.



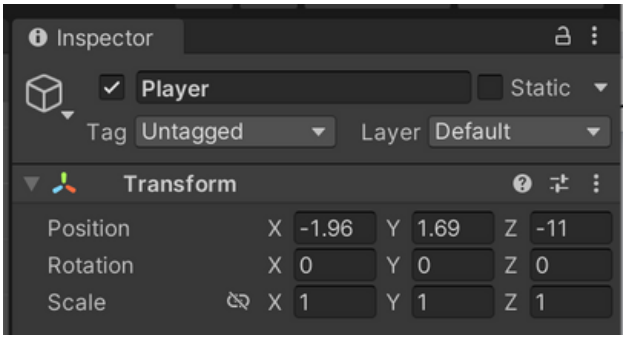
PROBLÈMES COURANTS AVEC LA CAMÉRA

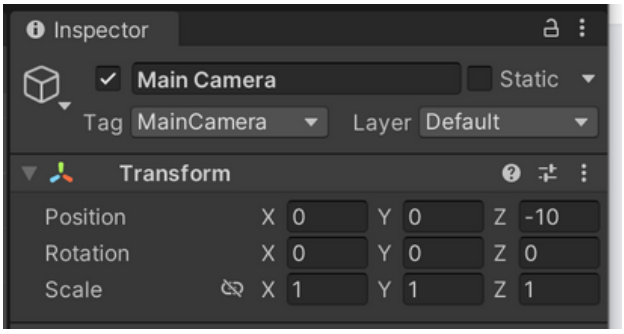
Il arrive qu'un Sprite que vous voyez dans la Scene ne s'affiche pas dans le jeu.



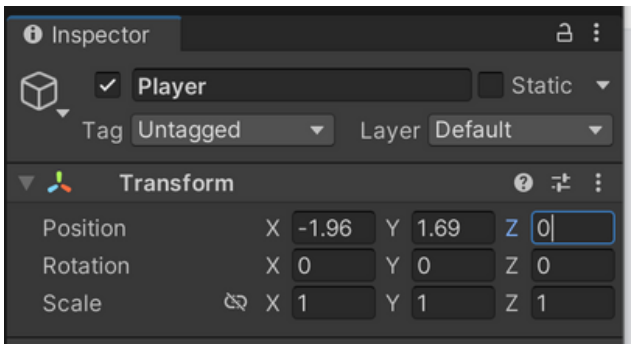
Ceci peut être dû à la position du Sprite par rapport à la caméra. Il faut garder en tête que même si vous faite un jeu en 2D, Unity est un moteur 3D. Les GameObjects sont donc positionnés dans un espace en trois dimensions. Il est possible donc que votre GameObject se trouve "derrière" la caméra.

Pour vérifier si c'est la cas, regardez les valeurs des z des positions, si la valeur du z de votre GameObject invisible est plus petite que celle de votre caméra alors celui-ci se trouve derrière elle.





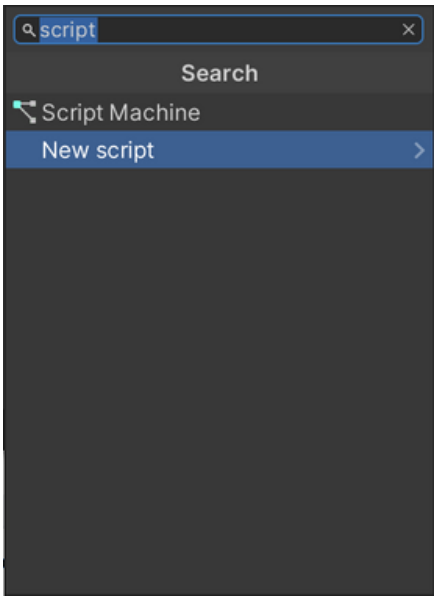
Il vous suffit alors de changer cette valeur dans le GameObject invisible pour qu'elle soit plus grande que celle de la caméra.

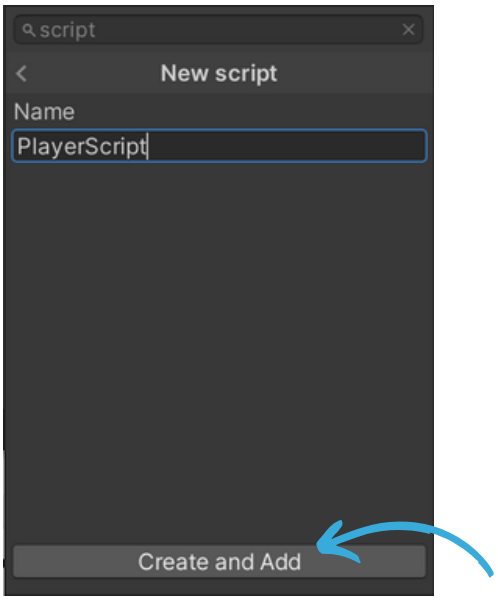


LES SCRIPTS

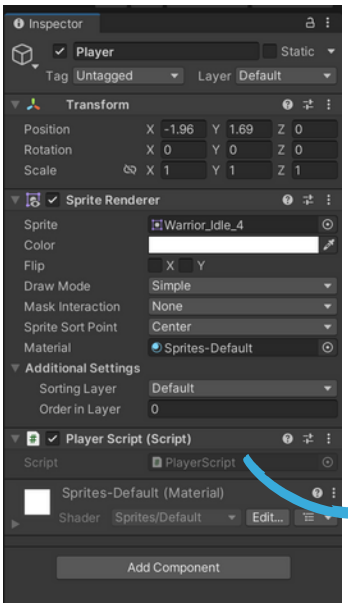
Il est possible de rajouter à un GameObject un composant de type Script. Ce dernier permet d'écrire des programmes en C# qui vont manipuler le GameObject. Nous allons voir plus en détail cela.

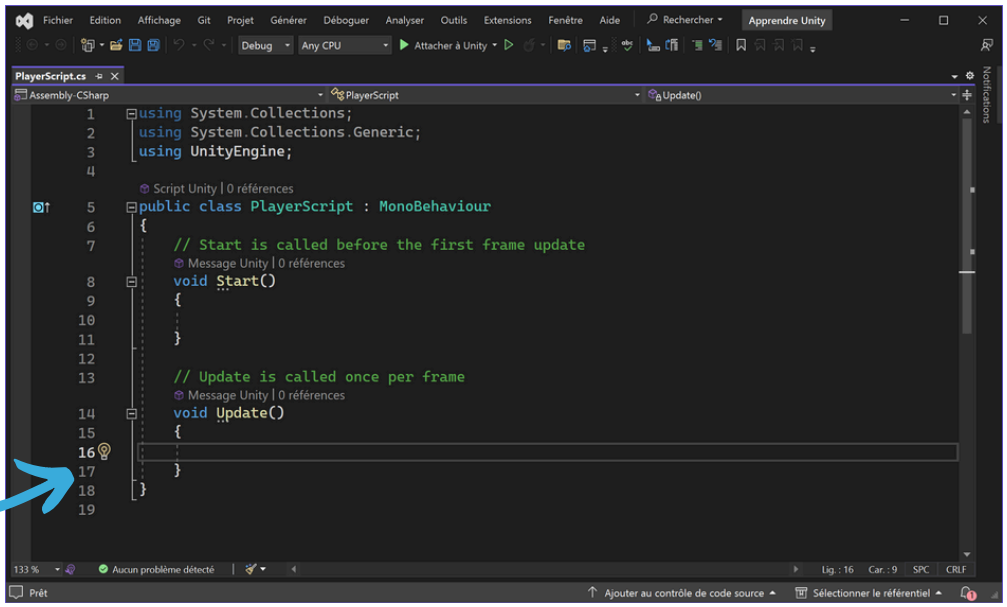
Tout d'abord, nous allons rajouter un composant Script à notre GameObject Player. Pour ce faire, cliquez sur "Add Component" et tapez script dans la barre de recherche. Cliquez ensuite sur "New script" et tapez le nom que vous voulez donner au Script de ce GameObject (ici PlayerScript) et cliquez sur "Create and Add".



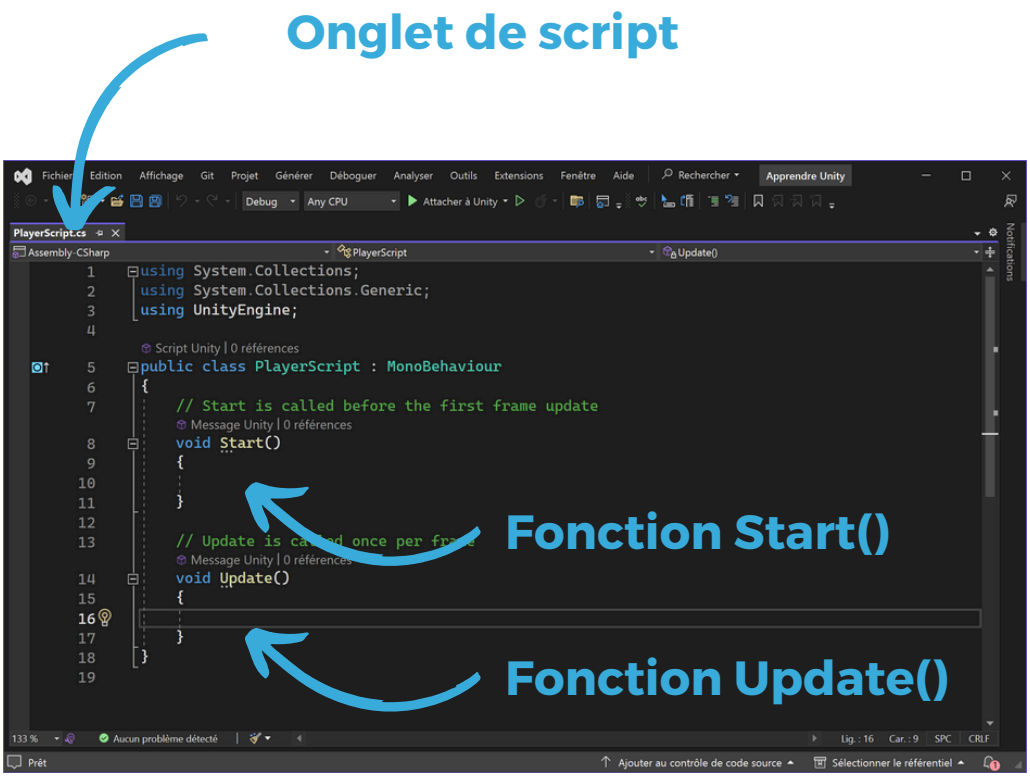


Vous verrez alors le composant Script qui se sera rajouté au GameObject. Double-cliquez sur le Script et une fenêtre Visual Studio s'ouvrira.





Nous n'allons pas nous attarder sur tous les éléments de la fenêtre Visual Studio mais nous allons nous intéresser à quelques uns pour commencer.



Les onglets
Visual Studio permet d'ouvrir plusieurs scripts en même temps, organisant ces derniers dans différents onglets dans lesquels il est possible de naviguer en cliquant sur l'un d'entre eux.

Start() est une fonction qui sera appelée une seule fois lorsque le jeu va démarrer.

Update() est une fonction qui sera appelée à chaque frame du jeu. Pour faire simple pour afficher ce que l'on voit à l'écran durant le jeu (une vidéo comme une autre), le programme du jeu va, à chaque instant, générer un certain nombre d'images qui seront affichées successivement pour créer le visuel du jeu.

La fonction Update() sera appelée lors de la création de chacune de ces images (à chaque frame).
Pour faire simple : cette fonction sera appelée un très grand nombre de fois par seconde durant le jeu.

Dans un Script nous pouvons utiliser toutes les instruction que l'on retrouve dans le langage C# en plus d'instruction spécifiques à Unity.

Les scripts permettent de manipuler toutes les valeurs des composants liés au GameObject auquel le Script est attaché. Nous allons voir ça tout de suite.

De plus, si vous avez correctement configuré votre projet, Visual Studio vous fera profiter d'un outil d'auto-complétions.

Nous allons commencé par faire un Script qui permet, lorsque le/la joueur/joueuse appuie sur la flèche de droite de son clavier, de faire avancer le personnage à droite.

Pour ce faire, nous allons devoir vérifier si le/la joueur/joueuse appuie sur la touche "Flèche de droite". Pour ça nous allons utiliser une structure **if** suivie de la condition **Input.GetKey(KeyCode.RightArrow)**.

```
void Update()
{
    if (Input.GetKey(KeyCode.RightArrow))
    {

    }
}
```

Le module **Input** comprend plein de fonction qui permettent de récupérer les entrée faites par le/la joueur/joueuse. La fonction **GetKey()** prend en paramètre le code de la touche à inspecter et renvoie True si la touche est pressée. Enfin, le **KeyCode.RightArrow** renvoie le code de la touche flèche de droite.

Input.GetKey(KeyCode.RightArrow) sera donc vrai si la touche flèche de droite est pressée à cette frame et faux si elle ne l'est pas. Le code dans la structure ne sera executé que si la touche est pressée.
Pour plus d'informations sur les fonction **GetKey** voir la fiche technique "Input.GetKey".

Nous allons donc devoir mettre dans le code sous la condition des instructions faisant bouger le GameObject vers la droite.

Pour ce faire nous allons utiliser l'instruction **transform.position += Vector3.right * Time.deltaTime** (en C# une instruction doit toujours être suivie de ;).

```
void Update()
{
    if (Input.GetKey(KeyCode.RightArrow))
    {
        transform.position += Vector3.right * Time.deltaTime;
    }
}
```

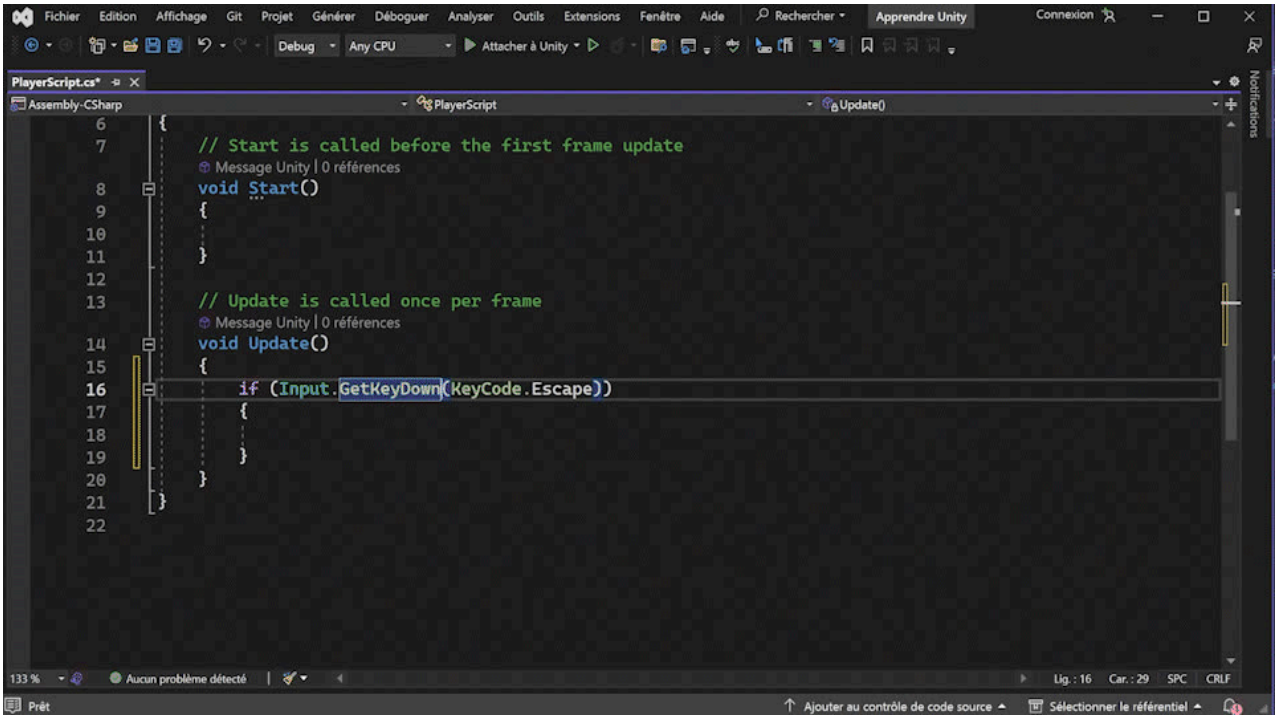
Pour bien comprendre ce qui va se passer ici, lorsque l'on va appuyer sur la flèche de droite, la variable **position** du composant **transform** du GameObject (qui est ici sous la forme d'un Vector3) sera incrémentée d'un vecteur dirigé vers la droite.

Comme la fonction **Update()** est appelée à chaque frame (et qu'il peut y avoir un grand nombre de frames par secondes) l'on va devoir normaliser cela pour obtenir un mouvement indépendant du nombre de frames.

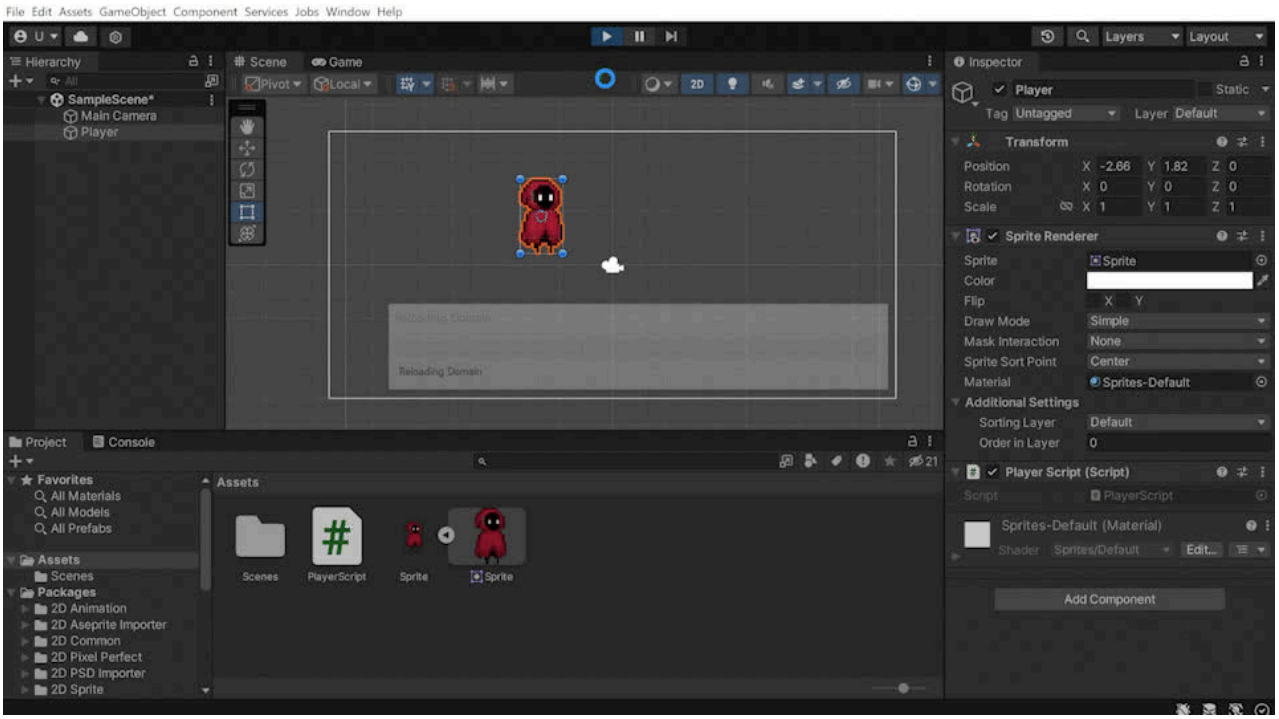
C'est pour cela que l'on multiplie le vecteur par **Time.deltaTime** qui est les temps écoulé depuis la dernière frame. Sauvegardez ensuite en appuyant sur les touches ctrl et s en même temps.

N'oubliez pas de sauvegarder votre programme quand vous avez fini de le modifier, sinon les modifications ne se répercuteront pas dans le jeu.

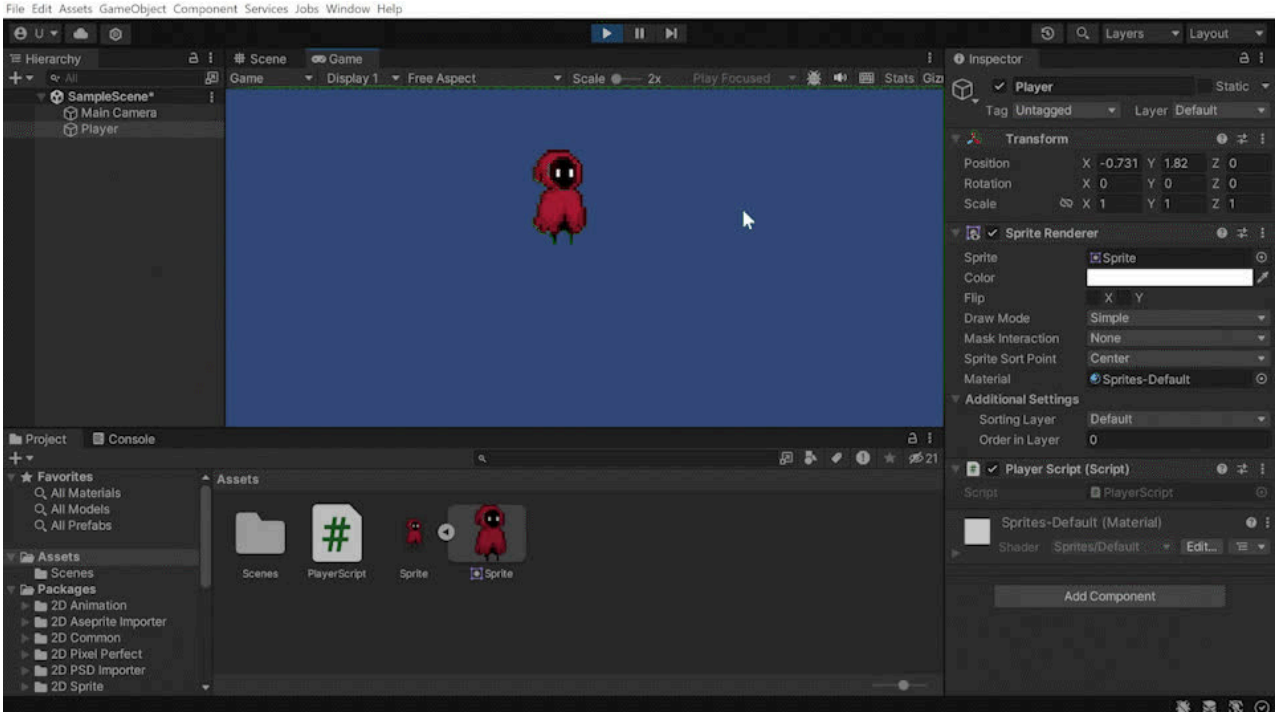
Avec l'auto-complétions tout ceci se fait rapidement :



Si nous revenons à la fenêtre Unity et que nous cliquons sur le bouton play en haut pour lancer notre jeu nous verrons un court temps de chargement puis la fenêtre Scene/Game va basculer sur l'onglet Game.



Un fois le jeu lancé, si vous appuyez sur la touche "Flèche de droite" vous verrez notre personnage se déplacer vers la droite.

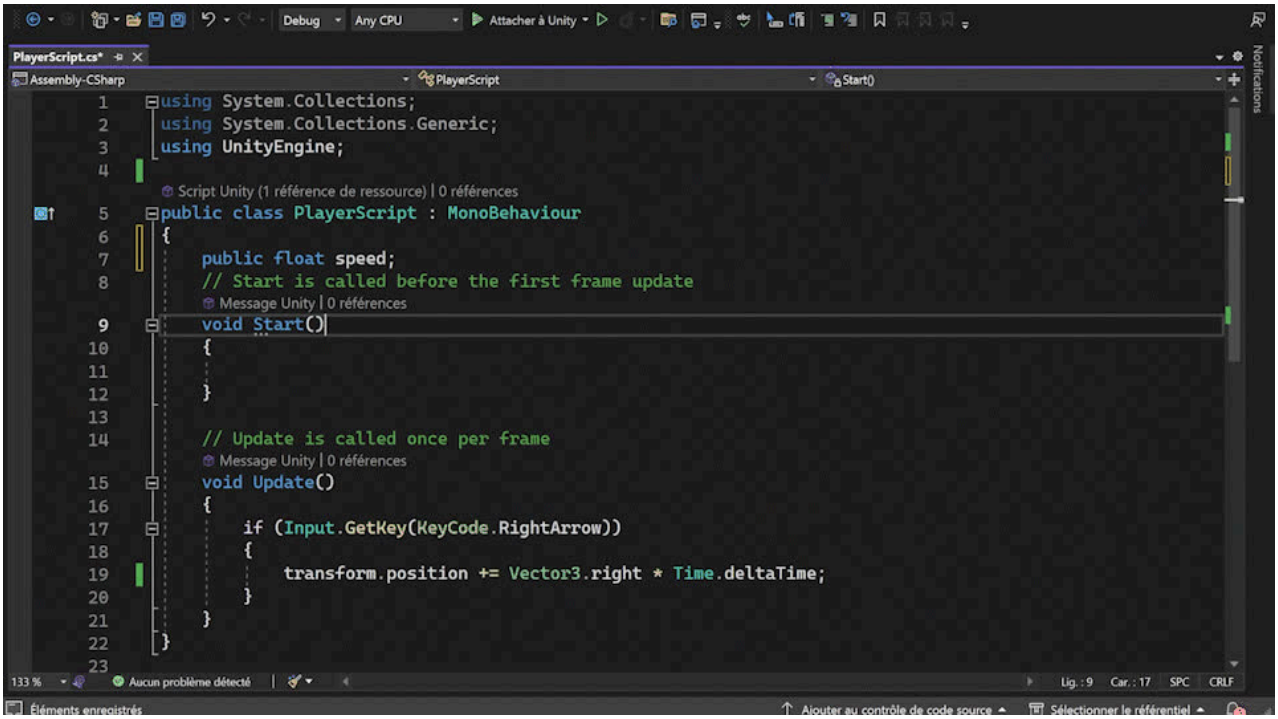


LES VARIABLES

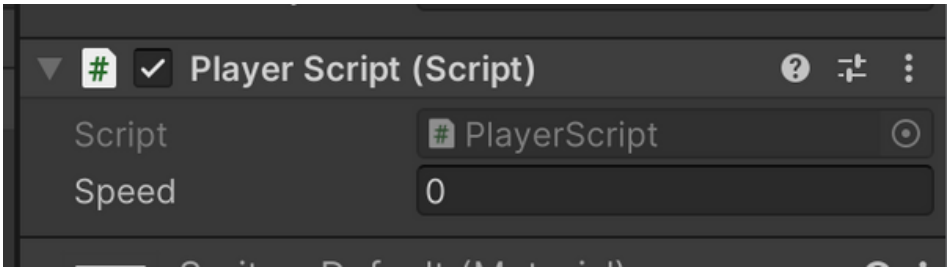
A l'étape précédente nous avons pu faire se déplacer notre sprite. Mais la vitesse de déplacement est probablement un peu lente, nous pourrions la changer en multipliant le **Vector3.right** pour augmenter le déplacement. Il nous faudra compiler et tester pour voir si la vitesse est bonne avec la nouvelle valeur mais cela risque de nous prendre pas mal de temps. Heureusement il existe un moyen de modifier une valeur dans le code via la fenêtre Unity durant l'exécution du jeu.

Pour ce faire nous allons déclarer une variable de type Float au début du script. Tapons donc **Public float speed;** Celle-ci doit se trouver entre l'accolade qui suit le **public class PlayerScript : MonoBehaviour** et le **void start()**. Et nous allons multiplier le **Vector3.right * time.deltaTime** par notre variable **speed**.

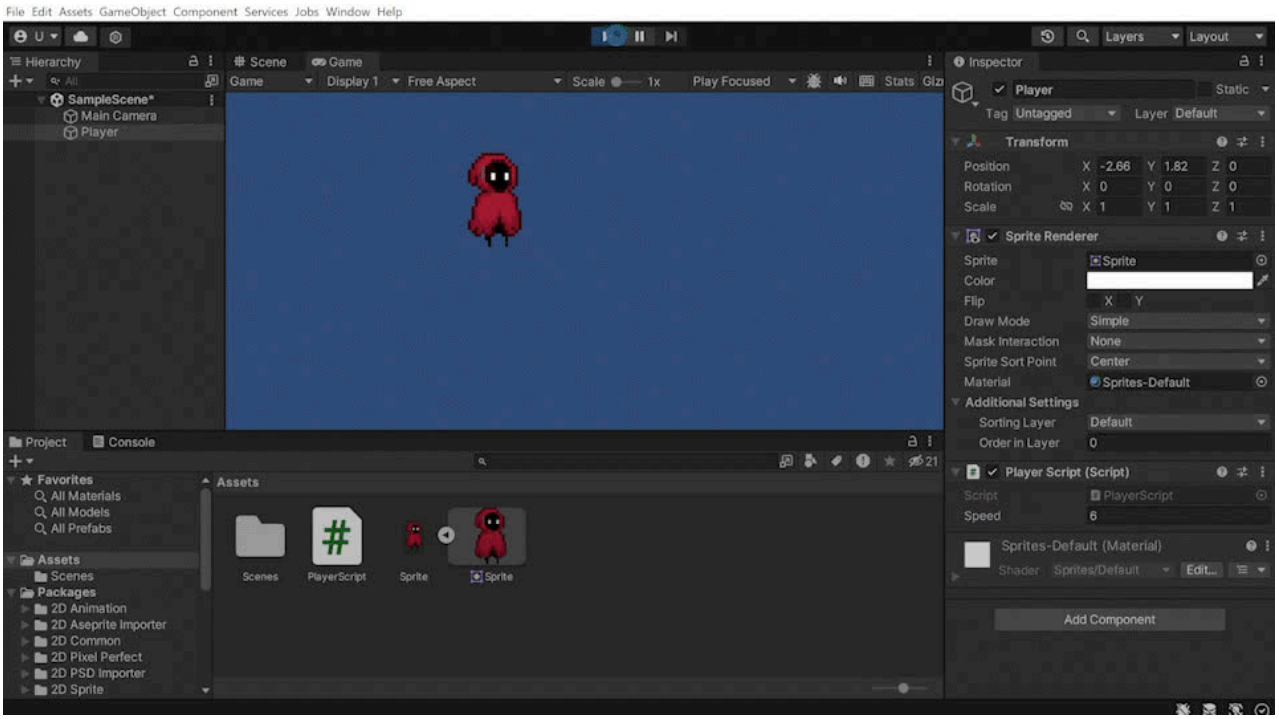
Il est aussi possible de déclarer des variable en **private** plutôt que **public**, dans ce cas la variable ne sera accessible que depuis l'intérieur de ce script (voir "Scope").



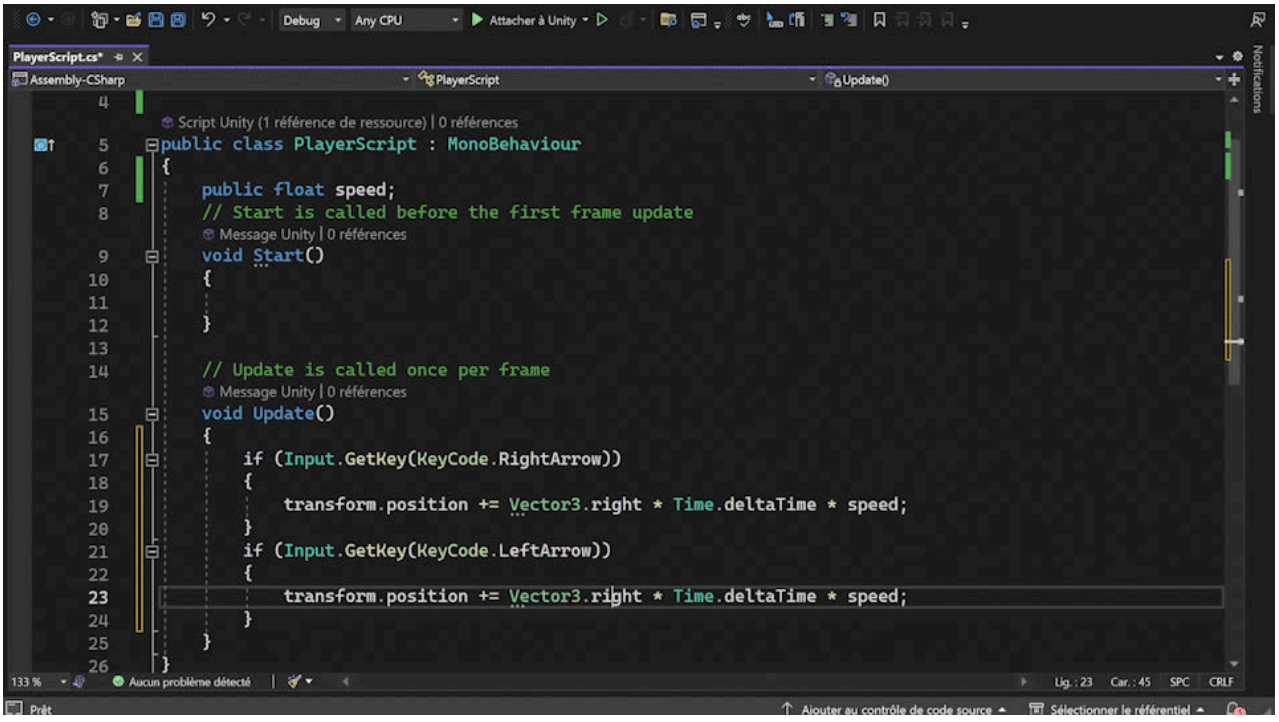
Si nous revennons à Unity nous verrons que notre variable Speed est visible dans le composant Player Script. Nous pouvons changer sa valeur ici et le changement sera répercuté lors de l'exécution du Script.



Si nous changeons cette valeur et lançons le jeu nous pouvons voir que la vitesse du personnage est plus rapide. L'utilisation de variables nous permettra plus tard de faire des ajustements de manière simplifiée.



Il nous faut maintenant recopier les instruction que l'on a écrites pour le déplacement vers la droite et les modifier pour faire un déplacement vers la gauche. Attention de ne pas oublier de changer à la fois la touche dans la condition et le vecteur.

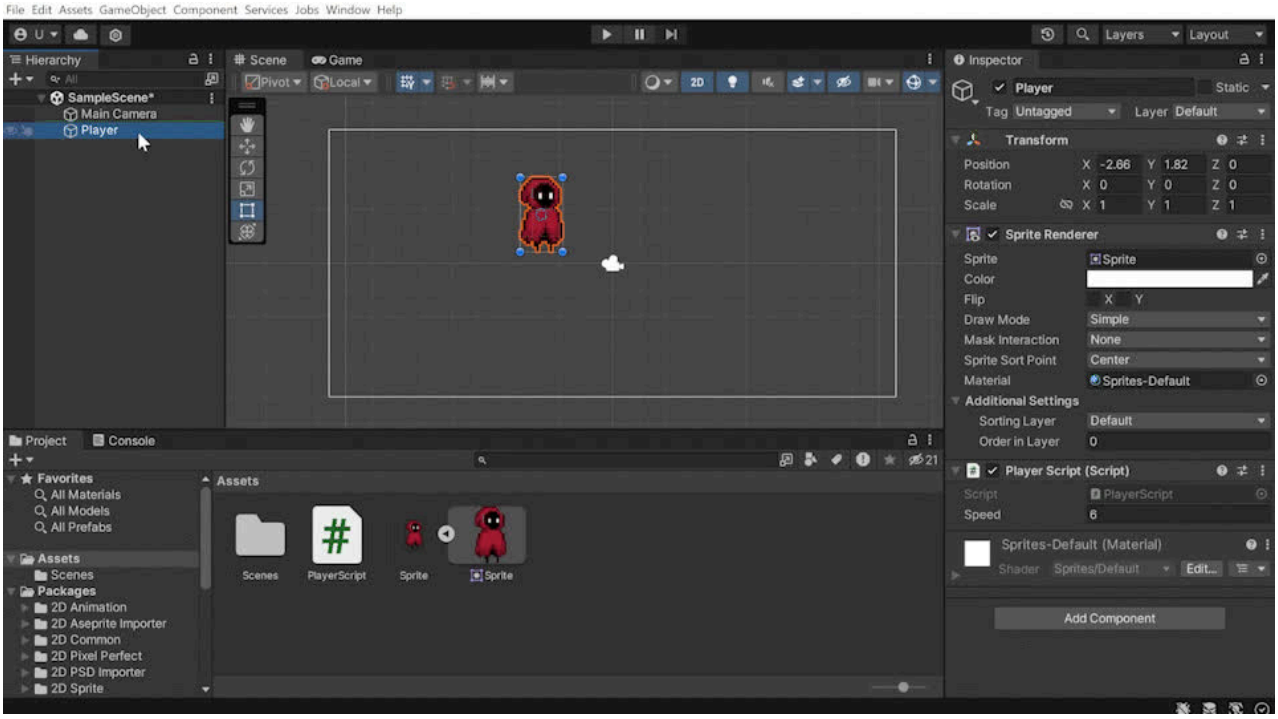


LA PHYSIQUE AVEC LES RIGIDBODY

La prochaine étape dans la création de notre jeu de plateforme serait de rajouter un effet de “gravité” à notre personnage. Nous pourrions rajouter des instructions dans le PlayerScript mais il y a plus simple.

Il existe parmi tous les composants inclus de base un composant spécialisé dans la simulation de phénomènes physiques. Il se décline en deux version : une version 3D et une 2D. Nous utiliserons ici la version 2D.

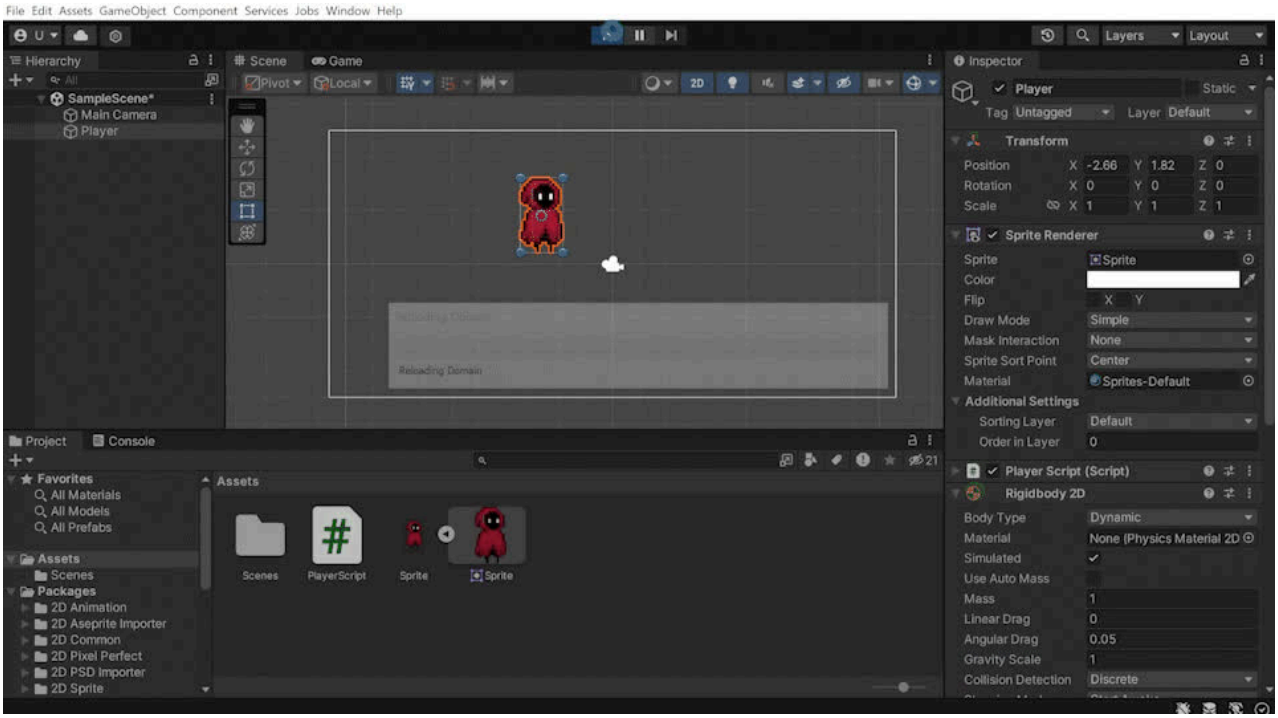
Le composant se dénome “RigidBody 2D”, pour le rajouter, comme d’habitude, cliquez sur le GameObject auquel vous voulez l’attacher et cliquez sur “Add Component”. Tapez ensuite “Rigidbody” et sélectionnez le RigidBody 2D.



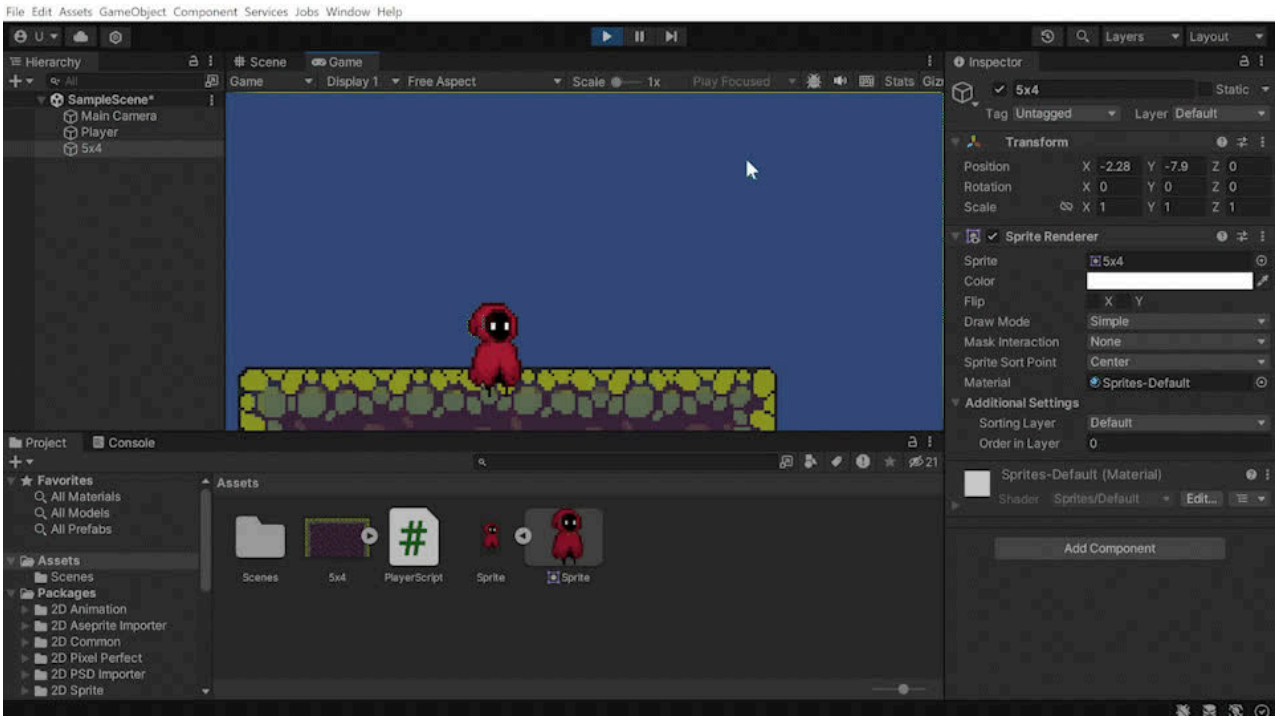
Comme vous pouvez le voir le composant RigidBody 2D comprend beaucoup de paramètres et prend en charge plusieurs phénomènes physiques (gravité, frottements. Ceux qui nous intéressent pour le moment sont : le Body Type (voir RigidBody 2D) et le paramètre Gravity Scale.

Le Body Type doit être ici Dynamic et le Gravity Scale va déterminer l’intensité de la gravité simulée pour ce Game Object.

Si l'on démarre le jeu l'on va voir le GameObject de notre personnage tomber sous l'effet de cette gravité.



La prochaine étape de notre jeu de plateforme serait de rajouter des plateformes. Pour ça nous pouvons tout simplement importer des Sprites de plateforme et créer les GameObjects adéquats.



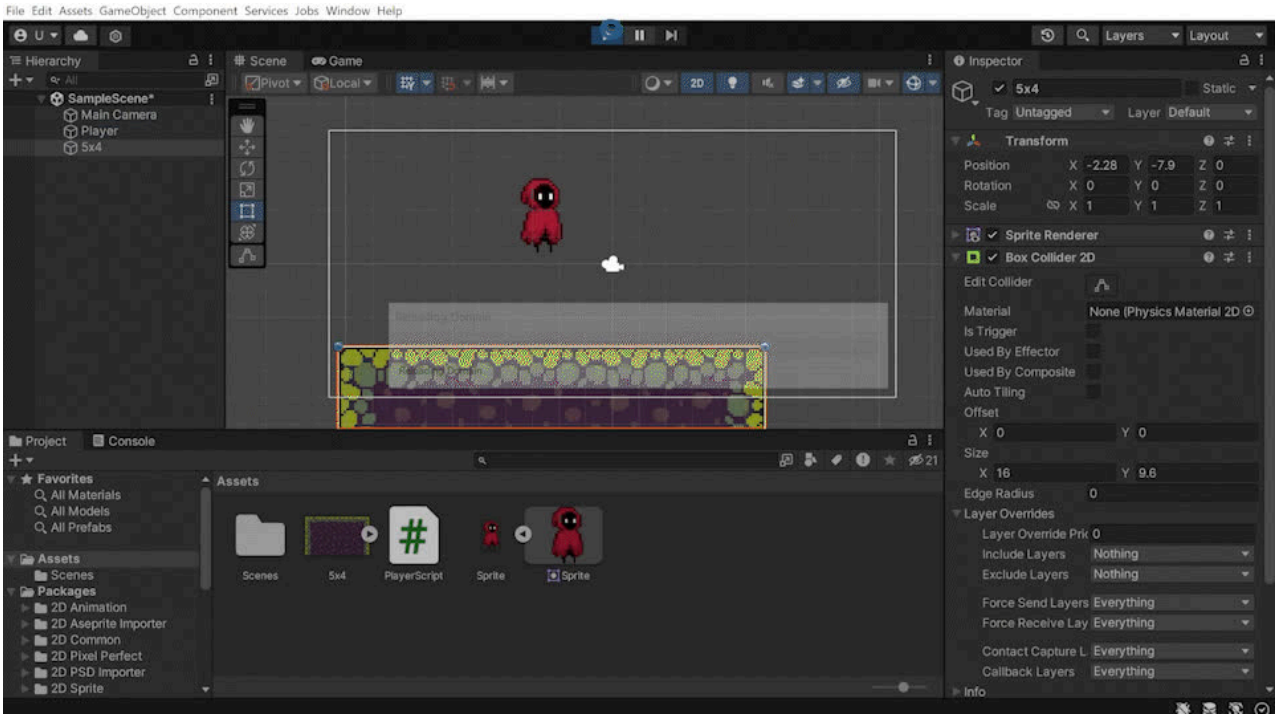
LES COLLISIONS AVEC LES BOXCOLLIDERS

Ah zut, notre personnage passe à travers la plateforme. Aucun composant ni aucune règle de simulation de la physique ne l'en empêche. De nouveau pour pallier à ce problème il existe un composant qui va permettre de gérer ce qu'on appelle les collisions entre les objets.

Il va nous falloir ajouter à tous les GameObjects qui ne peuvent pas se passer à travers un "Collider". Les composants de type Collider vont définir pour les objets auxquels ils sont attachés un volume (ou en 2D une surface) propre.

Les Colliders de deux objets ne pourront pas s'interpénétrer, les volumes ou surfaces définis par les Colliders formeront des sortes de volumes solides.

Un exemple vaut milles mots, voici pour notre jeu en ayant ajouté des BoxCollider 2D à notre personnage et à la plateforme :

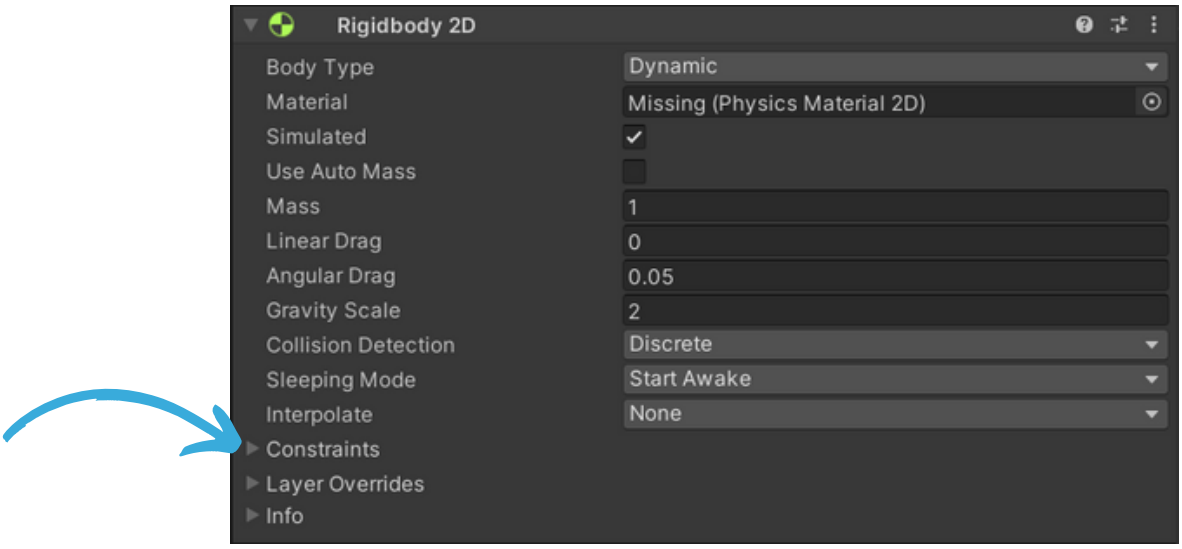


Nous pouvons voir notre personnage tomber puis s'arrêter dès qu'il a touché la plateforme. L'on peut ensuite le faire se déplacer avec les flèches gauches et droite.

Vous pourrez tomber sur un problème à cette étape, le personnage pourrait se mettre à tourner puis tomber si il se trouve au bord de la plateforme.

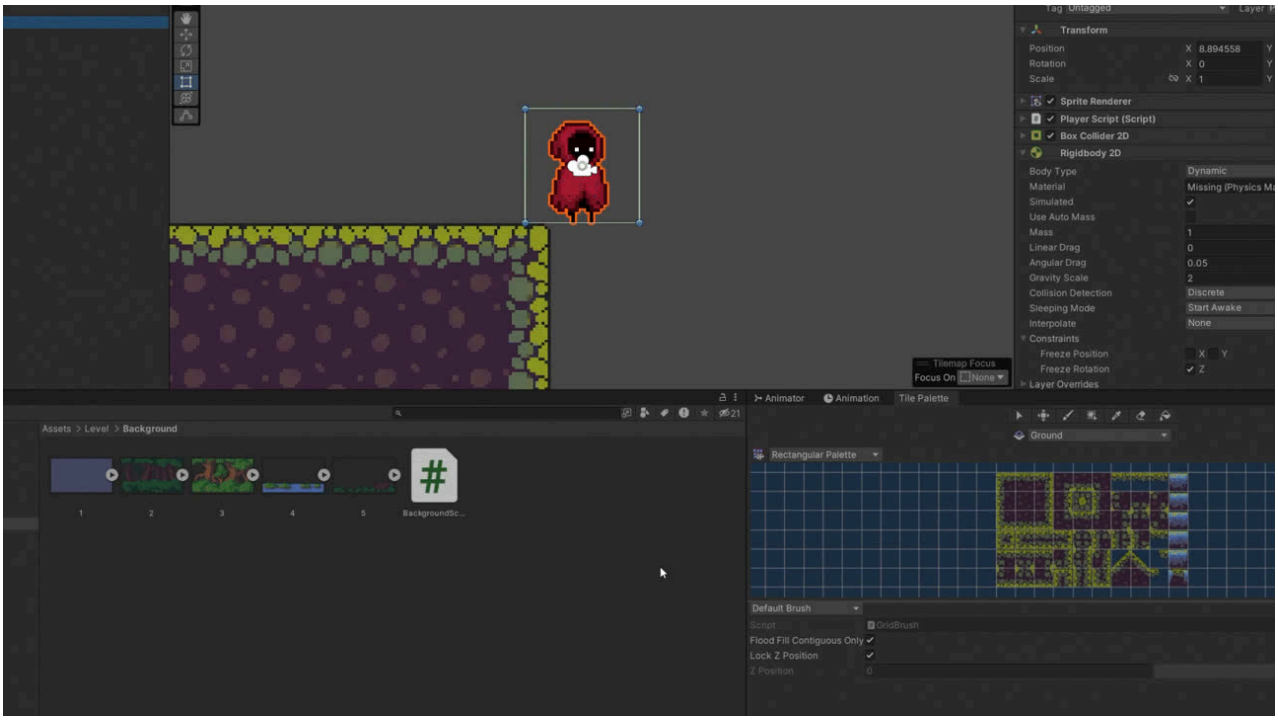
Ceci est plutôt logique, si vous poussez une tasse au bord d'une table elle va tomber en commençant à tourner. Mais ce comportement est rarement celui que vous voulez avoir pour un personnage dans un jeu de plateforme.

Pour régler ce problème il va falloir changer un paramètre du RigidBody2D. Ce paramètre se trouve dans le menu "Constraints", pour l'ouvrir cliquez sur "Constraints" dans le RigidBody.



Vous verrez alors différents options, il vous faut cocher la “Freeze rotation Z” (bloquer la rotation autour de l’axe Z). Cette dernière va empêcher le GameObject de tourner autour de l’axe Z.

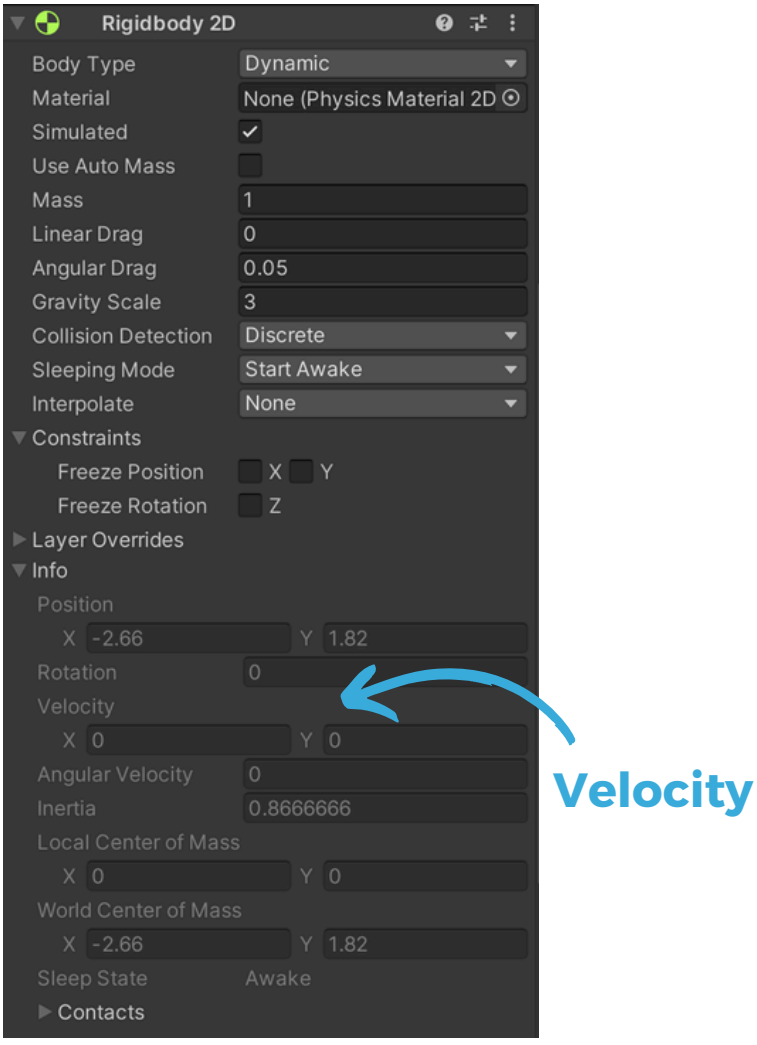
Si vous placez maintenant le personnage en bord de plateforme vous voyez qu’il ne tombe plus en tournant.



UTILISER LE RIGIDBODY DANS LE SCRIPT

Maintenant pour faire un jeu de plateforme digne de ce nom il faudrait que notre personnage puisse sauter. Si l'on réfléchit un peu quand on saute on se donne une impulsion vers le haut et puis la gravité va nous faire retomber sur le sol. Nous allons donc pouvoir utiliser le Rigidbody2D pour simuler cela.

Les Rigidbody disposent de nombreuses propriétés et celle qui va nous intéresser ici est la "Velocity" (littéralement "Vitesse"), elle correspond à la vitesse actuelle du Rigidbody. En 2D elle se présente sous la forme d'un Vector2 (représenté dans l'inspecteur par ses composantes x et y.

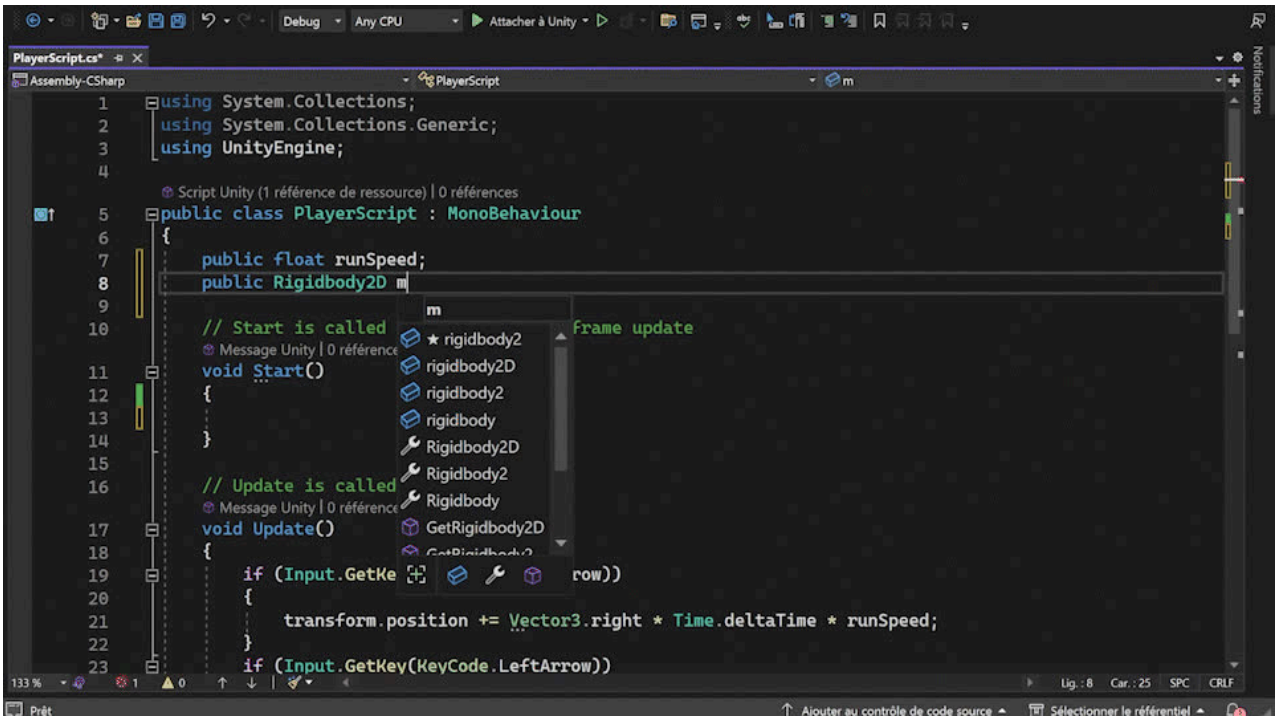


Velocity

L'idéal serait que quand le/la joueur/joueuse appuie sur la flèche du haut la composante y de la vélocité soit mise à une certaine valeur positive (pour sauter vers le haut). Pour ça nous allons retourner dans le programme du personnage.

Nous allons ici vouloir accéder à un composant que nous avons rajouté à notre GameObject, pour cela nous allons devoir le référencer dans notre programme. Commençons par créer une variable de type RigidBody2D.

De retour dans l'inspecteur nous pouvons retrouver cette variable, il nous suffit alors de glisser-déposer le Rigidbody2D dans la variable.



Nous allons maintenant pouvoir accéder aux propriétés du Rigidbody dans le programme en utilisant la variable dans laquelle nous l'avons référencé.

Pour ce faire il nous suffit de partir du nom de la variable suivi d'un point puis de la propriété à laquelle on veut accéder. Si l'on veut accéder à une sous propriété il faut spécifier tout le chemin avec les étapes séparées de points.

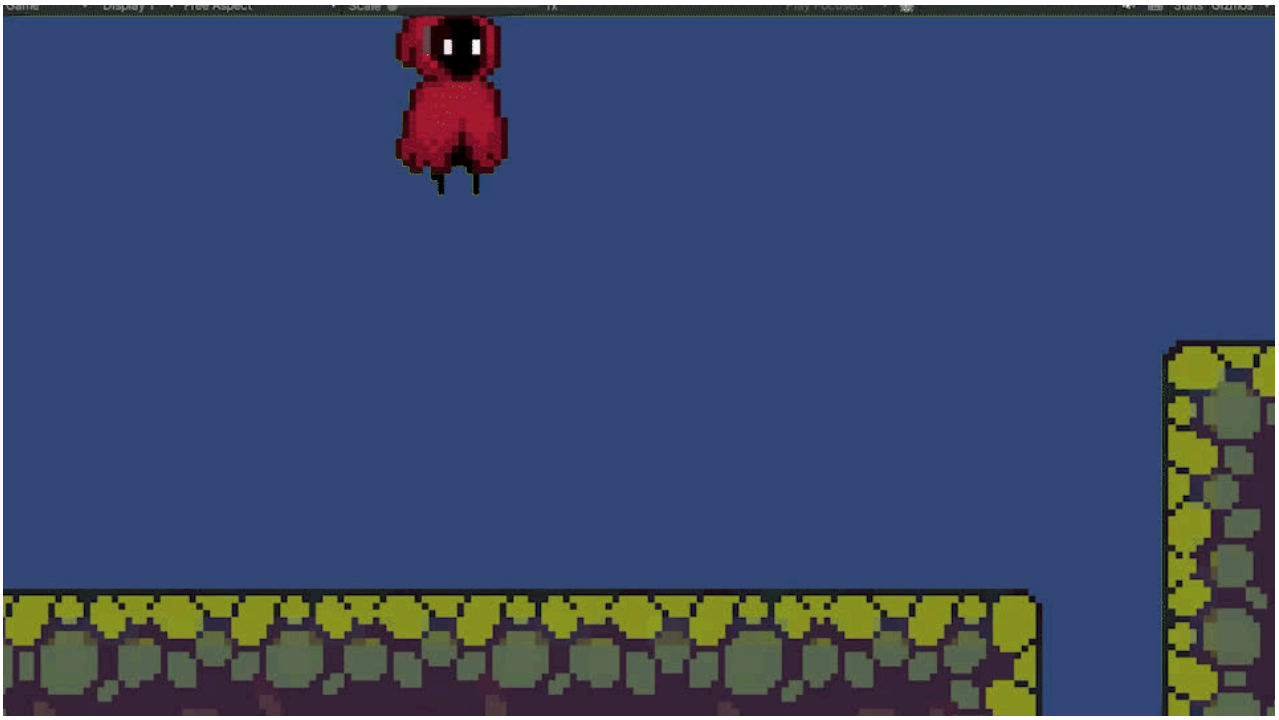
Ici l'on veut accéder au Rigidbody et plus spécifiquement à la propriété Velocity. Et l'on va vouloir ajouter à ce vecteur une impulsion vers le haut. Pour pouvoir régler la puissance du saut nous allons utiliser une autre variable de la même manière qu'avec la vitesse de déplacement.

```
if (Input.GetKeyDown(KeyCode.UpArrow))
{
    myBody.velocity += Vector2.up * jumpPower;
}
```

Vous noterez que nous avons dû utiliser ici un Vector2 plutôt qu'un Vector3. Ceci est dû au fait que la propriété Velocity d'un Rigidbody2D est un Vector2. De plus il faut maintenant utiliser Input.GetKeyDown (voir Input.GetKey).

Si nous revenons au jeu nous pouvons voir que le saut fonctionne bien.

Un problème subsiste cependant, si nous appuyions plusieurs fois de suite sur la flèche du haut notre personnage saute plusieurs fois même s'il ne touche pas le sol.

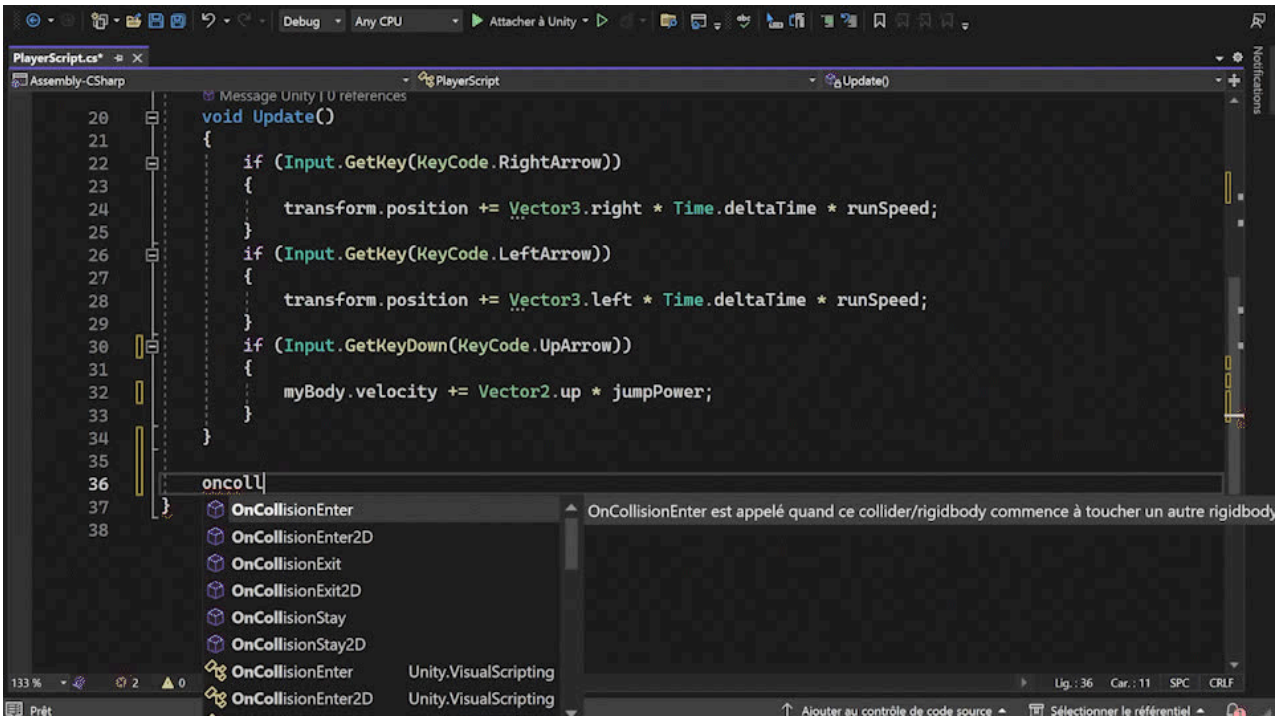


Pour régler ce problème il faudrait vérifier si notre personnage touche le sol et l'autoriser à sauter que si il touche effectivement le sol.

Pour ce faire nous allons utiliser la fonction **OnCollisionEnter2D**, nous allons la déclarer après la fonction **update**. Cette fonction sera appelée à chaque fois que le Collider2D entrera en collision avec un autre Collider.

OnCollisionEnter2D() est une fonction similaire à **Start()** et **Update()** dans le sens qu'on y mettra des choses qui devront être exécutées à certains moments.
Cette fonction sera appelée à chaque fois que le Collider du GameObject auquel le script est attaché entre en collision avec un autre Collider.

Il nous faut ausi une variable booléenne qui va garder en mémoire le fait que le personnage puisse sauter ou non. Cette dernière sera définie à True quand le personnage touche le sol et à False quand il sautera.



Pour bien comprendre le programme, quand le Collider entre en collision (**OnCollisionEnter2D**) la variable **canJump** est définie à **True**.
Si on appuie sur la flèche du haut et que la variable **canJump** est **True** alors seulement dans ce cas là le personnage saute et la variable **canJump** est définie à **False**, ce qui rend impossible un autre saut tant que le Collider n'entre pas en collision.

Attention car la variable canJump est mise à True pour n'importe quelle collision, ce qui peut nous poser problème. Pour mettre en place un système plus robuste il faudrait utiliser un Raycast (voir fiche du même nom).

MOUVEMENTS DE CAMÉRA

Il faudrait maintenant que la caméra suive notre personnage. Pour cela il faut se rappeler que même la caméra est un GameObject et que l'on peut donc lui assigner un composant Script qui s'occuper de lui faire suivre le personnage.

```
public class CameraScript : MonoBehaviour
{
    public Transform playerTransform;

    Message Unity | 0 références
    void Update()
    {
        transform.position = new Vector3(playerTransform.position.x, playerTransform.position.y, -10);
    }
}
```

L'idée ici est de récupérer le composant Transform du personnage et de placer la caméra à la même position, à l'exception de l'axe Z où la caméra doit rester en -10. C'est pourquoi à chaque Frame nous changeons la position de la caméra et la mettons à un nouveau Vector3 qui à les mêmes coordonnées que le personnage, sauf pour l'axe Z qui est à -10.

RÉAPPARITION

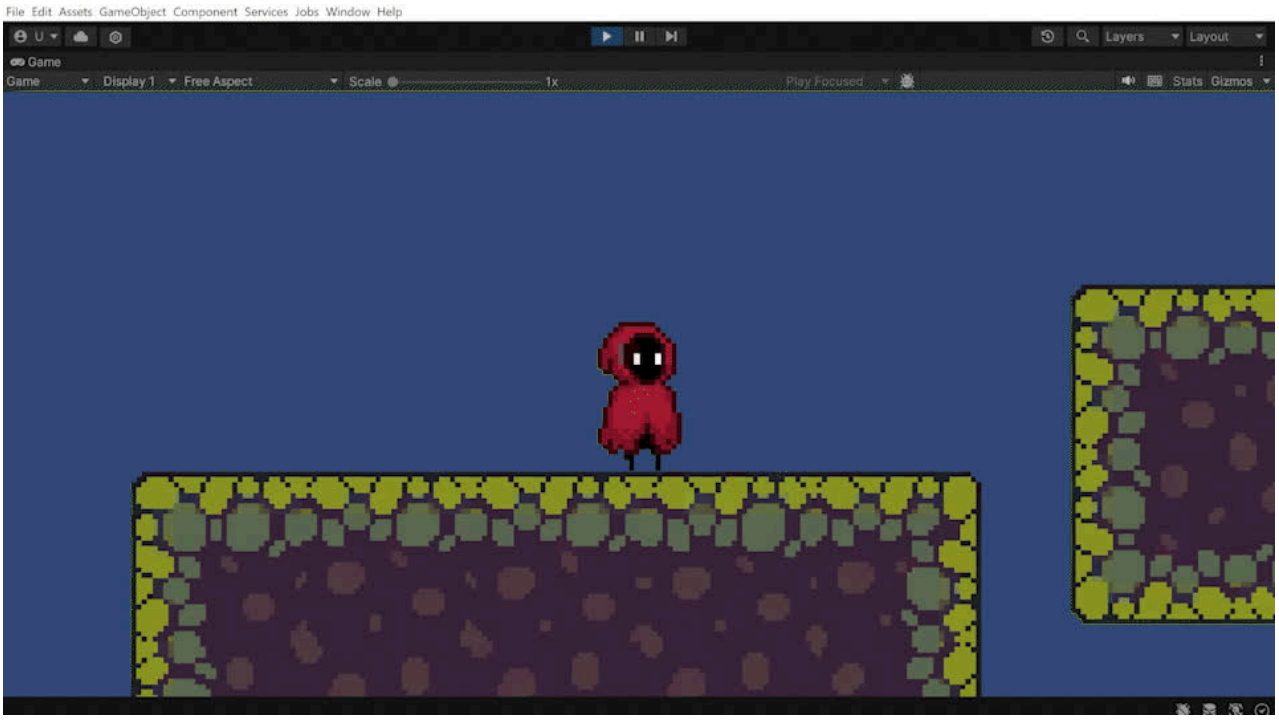
Nous devons régler un petit problème, si notre personnage tombe dans un trou, rien ne se passe et le joueur se retrouve bloqué en dessous du niveau que nous avons créé. Il faudrait que quand il tombe il réapparaisse au début du niveau.

Pour ce faire nous allons modifier un peu le programme du personnage joueur. Nous allons rajouter quelques lignes qui vont à chaque frame vérifier si le personnage joueur est en-dessous d'une certaine hauteur.

Si le personnage passe en-dessous de cette hauteur il sera "téléporté" au début du niveau et pourra recommencer.

```
if (transform.position.y < -10)
{
    transform.position = new Vector3(-3, 0, 0);
}
```

Ce qui donnera ceci :



Notez que l'on sera embêté si l'on décide de faire aller notre niveau vers le bas. Si notre personnage descend en-dessous de -10 de hauteur il sera systématiquement téléporté au début.

Mais rien ne nous empêche de mettre une autre condition sous laquelle notre personnage réapparaitre. Il suffira de changer la condition après le **if**.

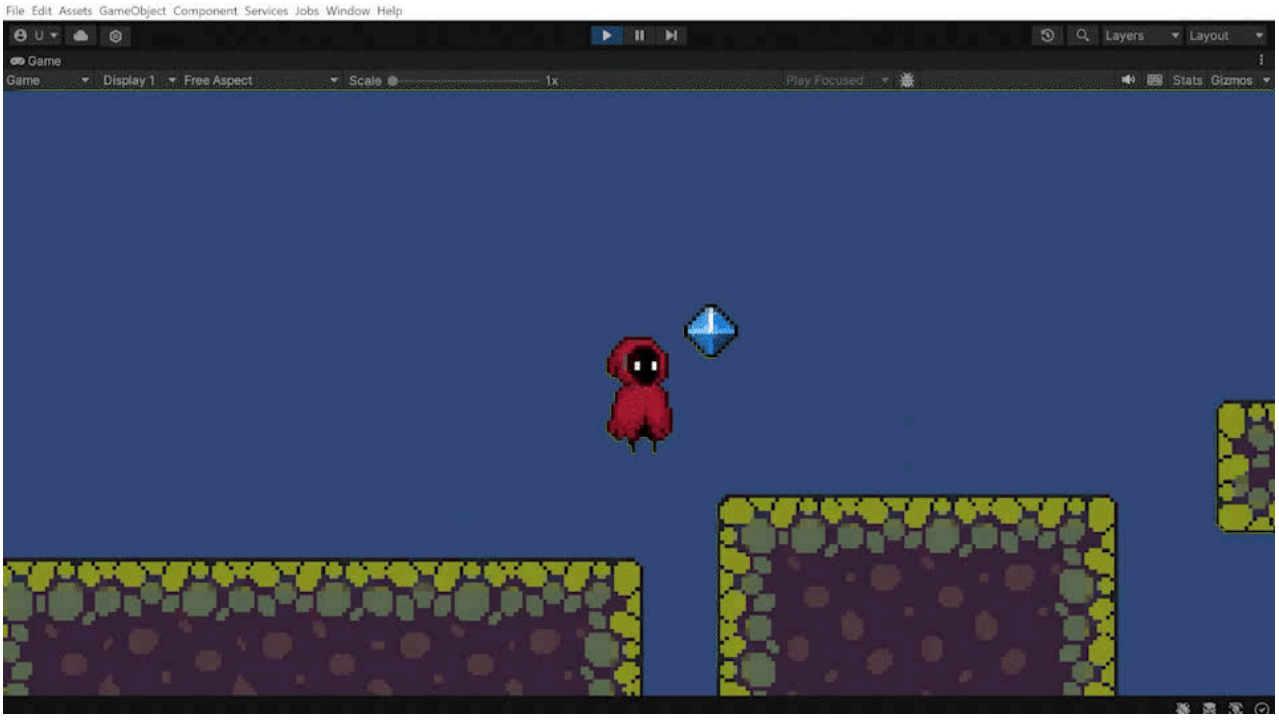
OBJETS INTERACTIFS

Objectifs :
- Créer des objets qui interagiront avec le joueur

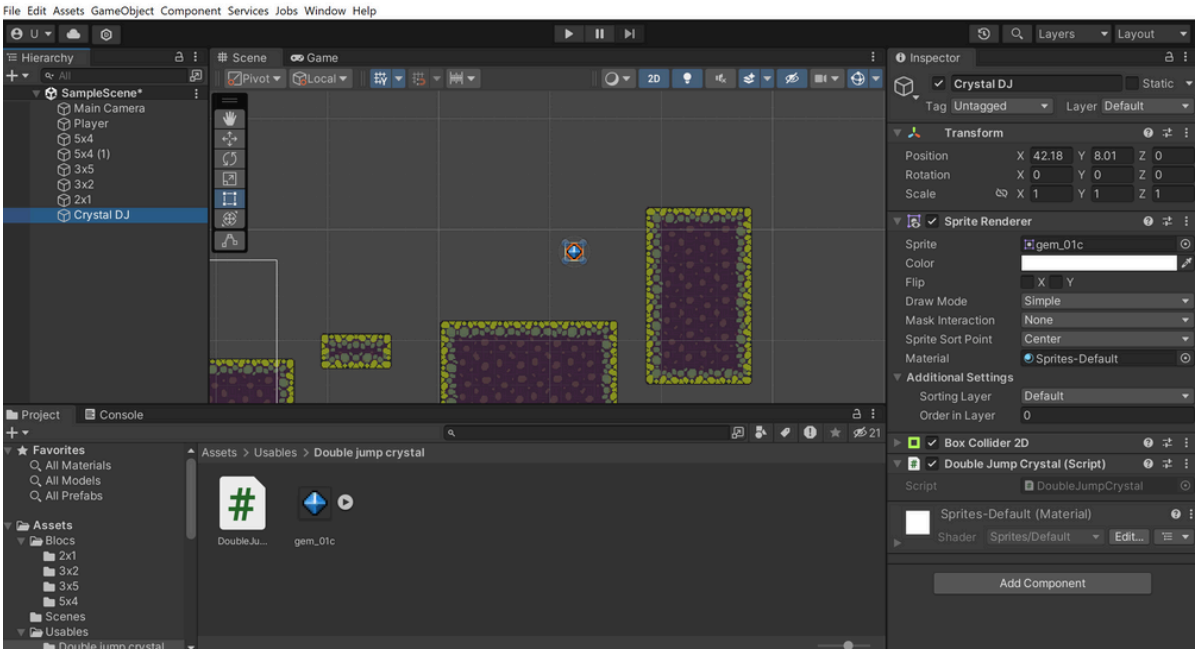
Déroulement de l'activité

Introduction	• Découverte de l'interface collectivement. • Rechercher les différentes zones de travail, ...
Exploration	• Former des binômes, distribuer les fiches et les lancer sur le défi 1. • Les élèves explorent librement l'environnement. Ils n'ont pas de consignes précises mais les inviter à observer les différentes parties de l'interface de programmation.

Nous allons maintenant voir comment créer des objets qui pourront interagir avec le personnage joueur. Nous allons créer un objet qui permettra au joueur de sauter une seconde fois quand il est dans les airs, ce qui lui était précédemment impossible.



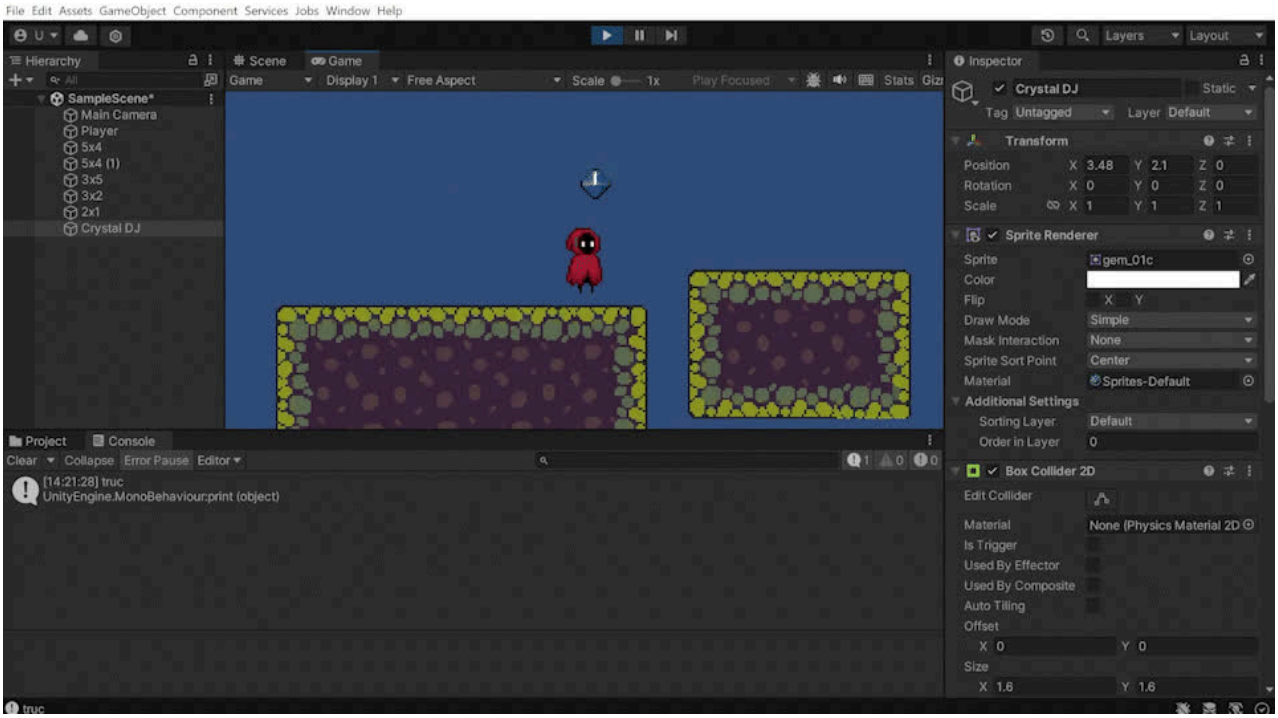
Nous allons d'abord commencer par créer un nouveau GameObject avec un nouveau Sprite, nous allons lui donner un BoxCollider2D et un Script. Nous avons vu comment faire ça précédemment.



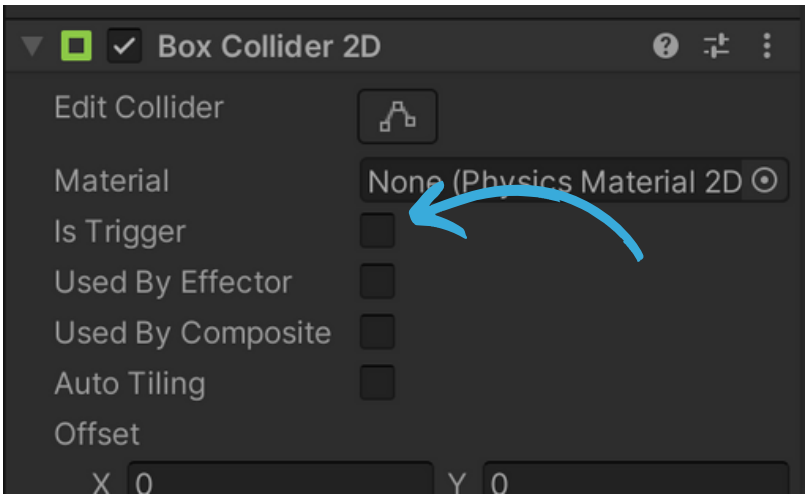
Maintenant nous voudrions qu'il se passe quelque chose quand le joueur touche la cristal. Pour ça nous allons utiliser la fonction la **OnCollisionEnter2D()** que nous avions déjà mis en place dans le script du joueur.

```
Message Unity | 0 références
private void OnCollisionEnter2D(Collision2D collision)
{
    canJump = true;
}
```

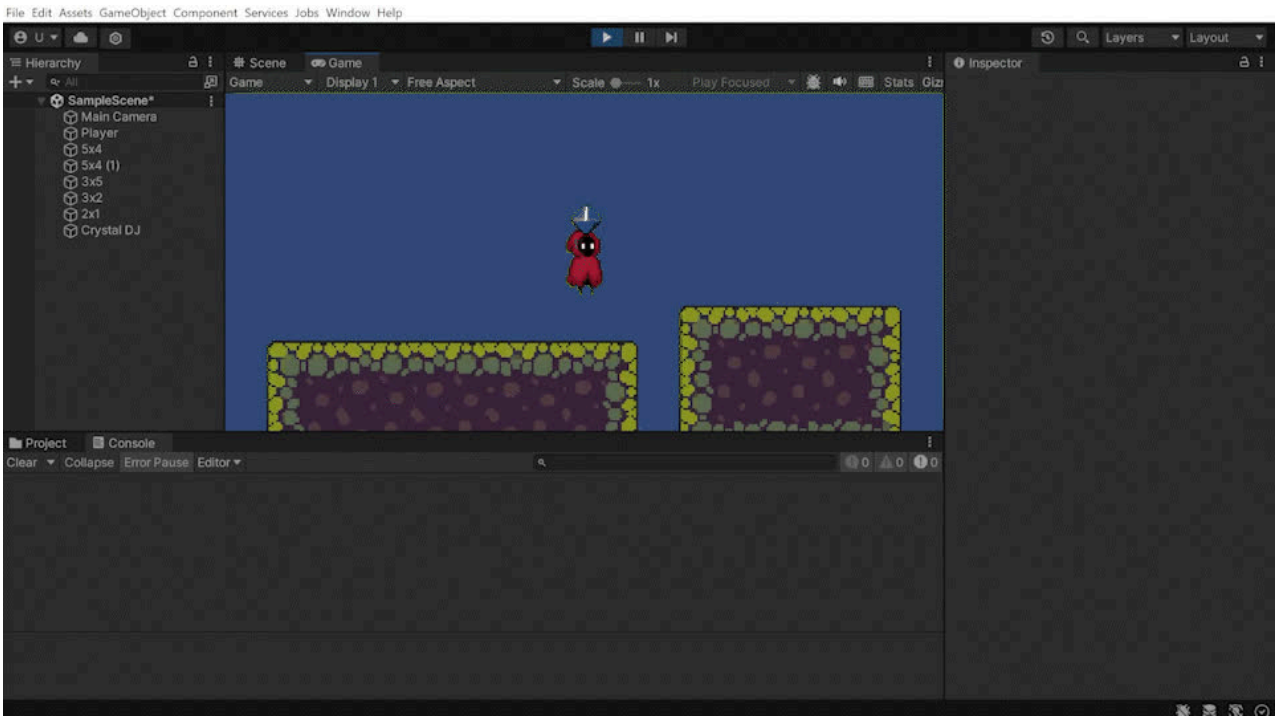
Dans le code juste au dessus nous avons la fonction **OnCollisionEnter2D()** qui se trouve à la suite des fonctions **Start()** et **Update()**, elle ne contient qu'un instruction qui assigne True à la variable **CanJump**. Ici quand le joueur touchera le cristal La fonction **OnCollisionEnter2D()** sera déclenchée et la variable sera bien assignée à True, donc le joueur pourra sauter de nouveau.



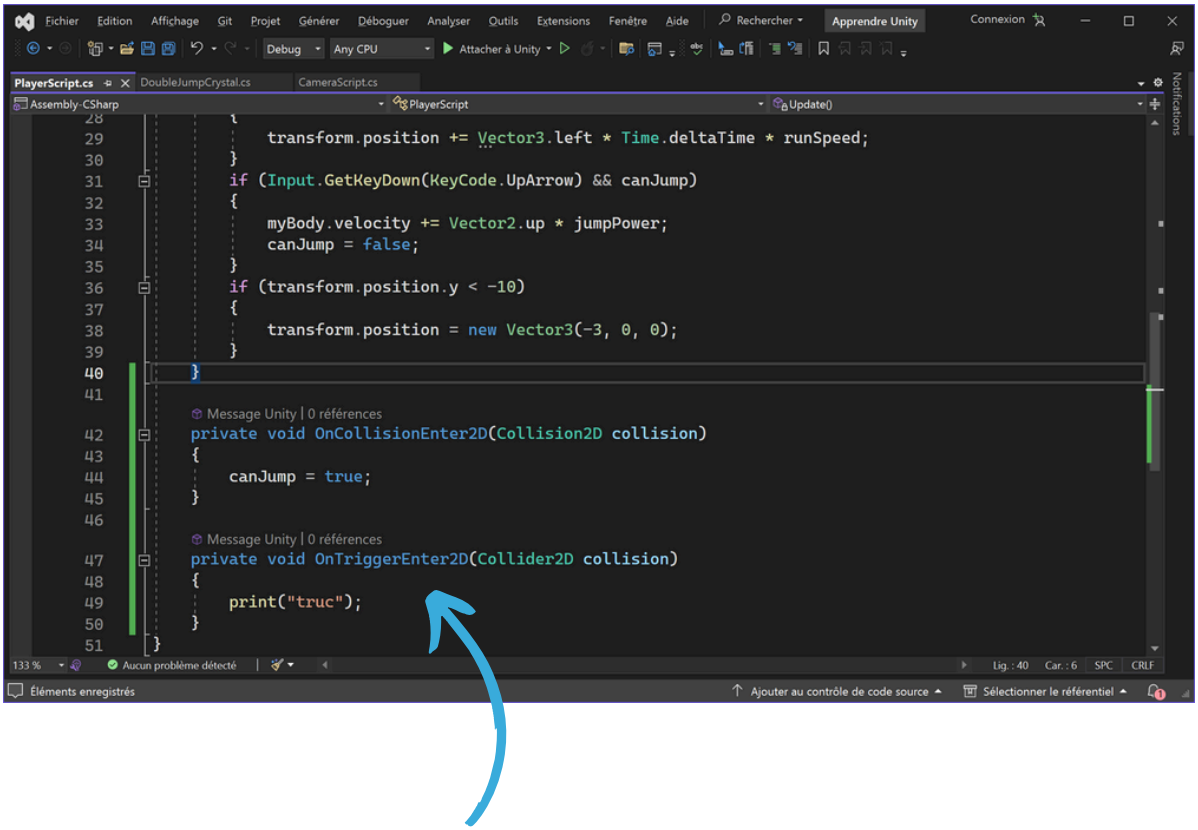
Nous pouvons voir qu'il y a un problème, vu que nous utilisons un Collider, notre personnage se cogne au cristal et ne passe pas à travers.
Pour permettre au personnage de traverser le cristal mais de quand même déclencher des instructions quand il le touche il nous faut légèrement changer le Collider. Il existe une option pour que ce dernier détecte les collisions sans faire obstacle, c'est le "Is Trigger".



Dans ce cas le BoxCollider2D va bien détecter les collision mais ne va pas bloquer le chemin.

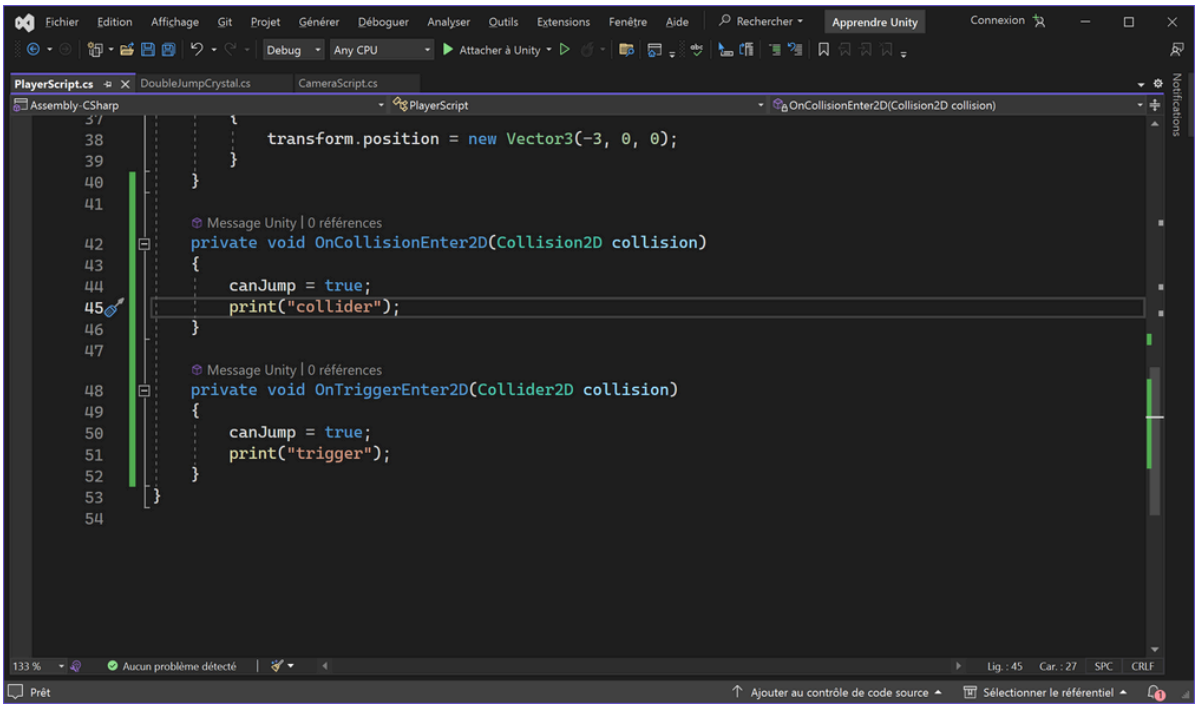


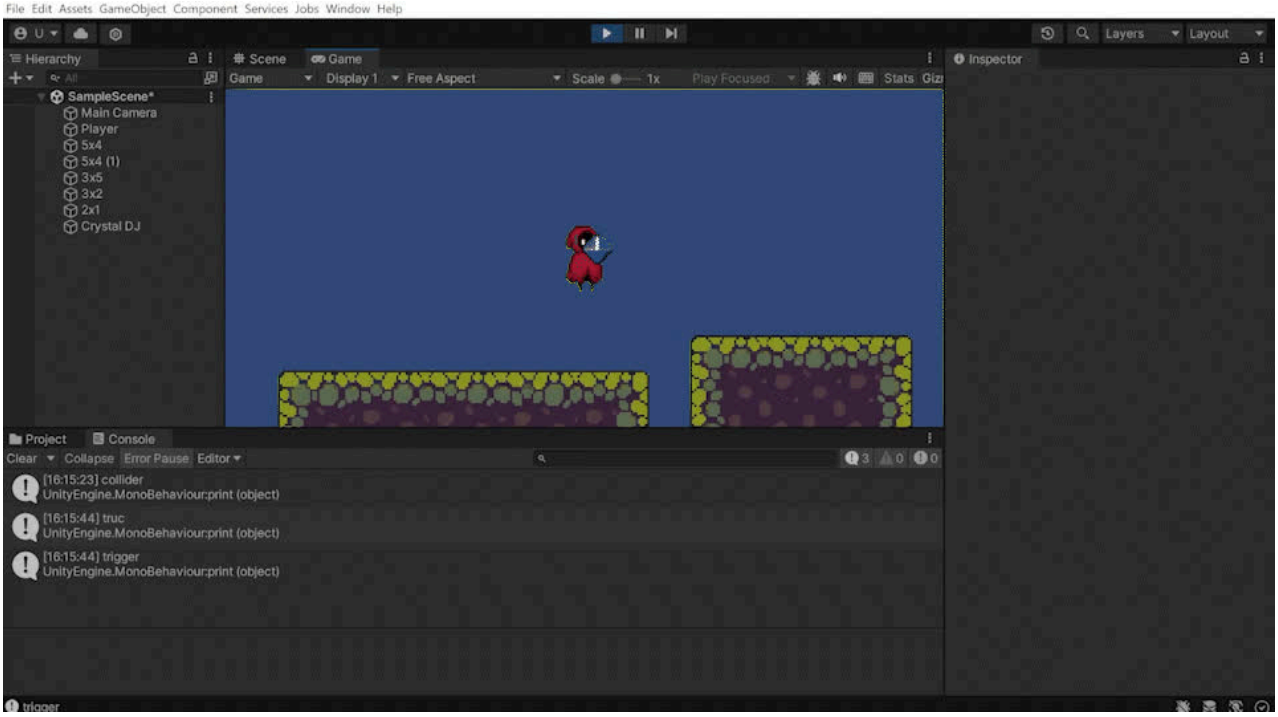
Ah zut, maintenant l'instruction print("truc") n'est plus déclenchée, vu qu'il n'y a pas de "collision" entre les deux BoxColliders la fonction **OnCollisionEnter2D()** n'est plus appelée. Il nous faut utiliser une autre fonction très similaire, **OnTriggerEnter2D()**.



Les Colliders dont l'option "Is Trigger" est cochée ne déclenchent pas la méthode OnCollisionEnter, ce qui est logique vu qu'ils ne déclenchent pas de collision. C'est pourquoi il faut utiliser la méthode OnTriggerEnter.

Nous pouvons simplement mettre dans cette fonction une assignation de la variable canJump à True (dans le code qui suit il y a deux “print” en plus pour permettre de bien comprendre ce qui va se passer).





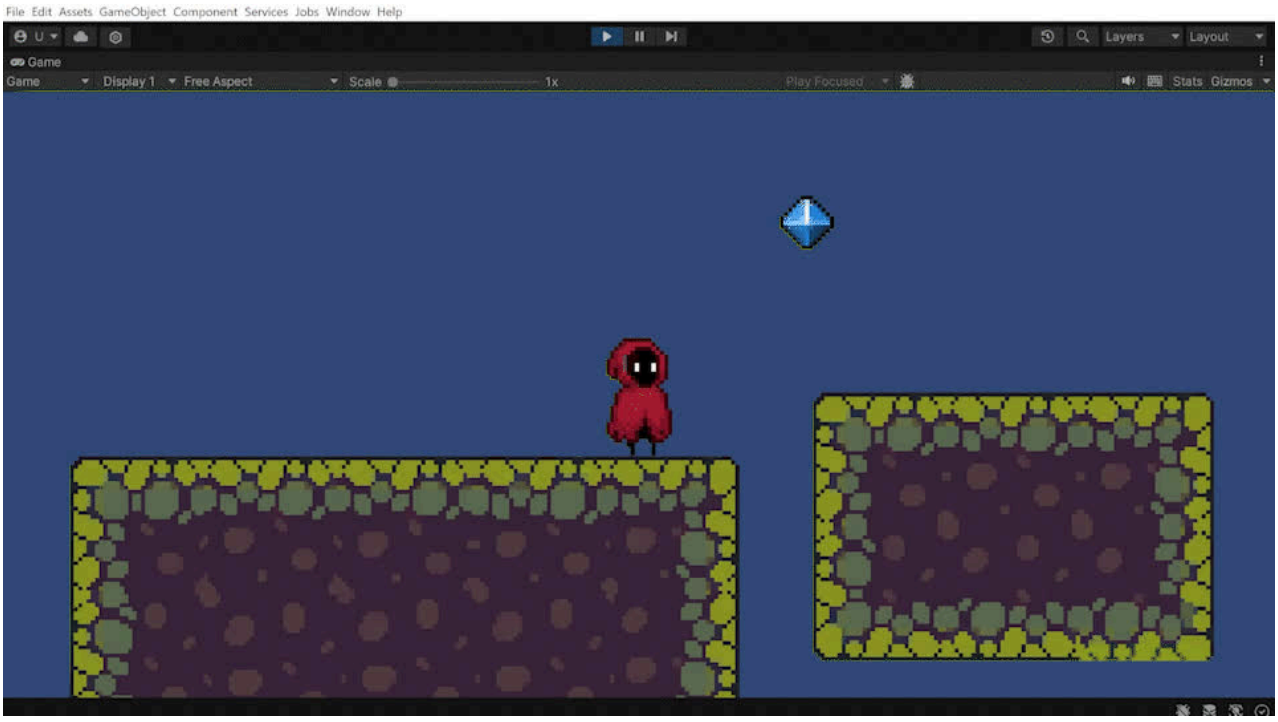
Dans l’extrait ci-dessus j’ai pu sauter une première fois puis une seconde fois quand mon personnage a touché le cristal.

Maintenant nous allons nous attacher à améliorer l'esthétique du cristal et de son fonctionnement. L'esthétique, les visuels, les son, etc ne sont vraiment pas à négliger dans un jeu vidéo. Ces détails participent à l'expérience de jeu et contribuent énormément à leurs qualités.

Nous allons commencer par quelques petites choses simples, nous allons donner un peu de mouvement au cristal en le faisant "léviter"

Pour ce faire nous allons rajouter dans son code quelques instructions qui lui feront aller de haut en bas. Je vous invite à regarder et bien comprendre le code qui suit. N'oubliez pas de donner des valeurs aux variables **speed** et **period** dans l'inspecteur. Le % dans le code ci-dessous est ce qu'on appelle en programmation un modulo, c'est une opération qui donne le reste de la division du premier nombre par le second.

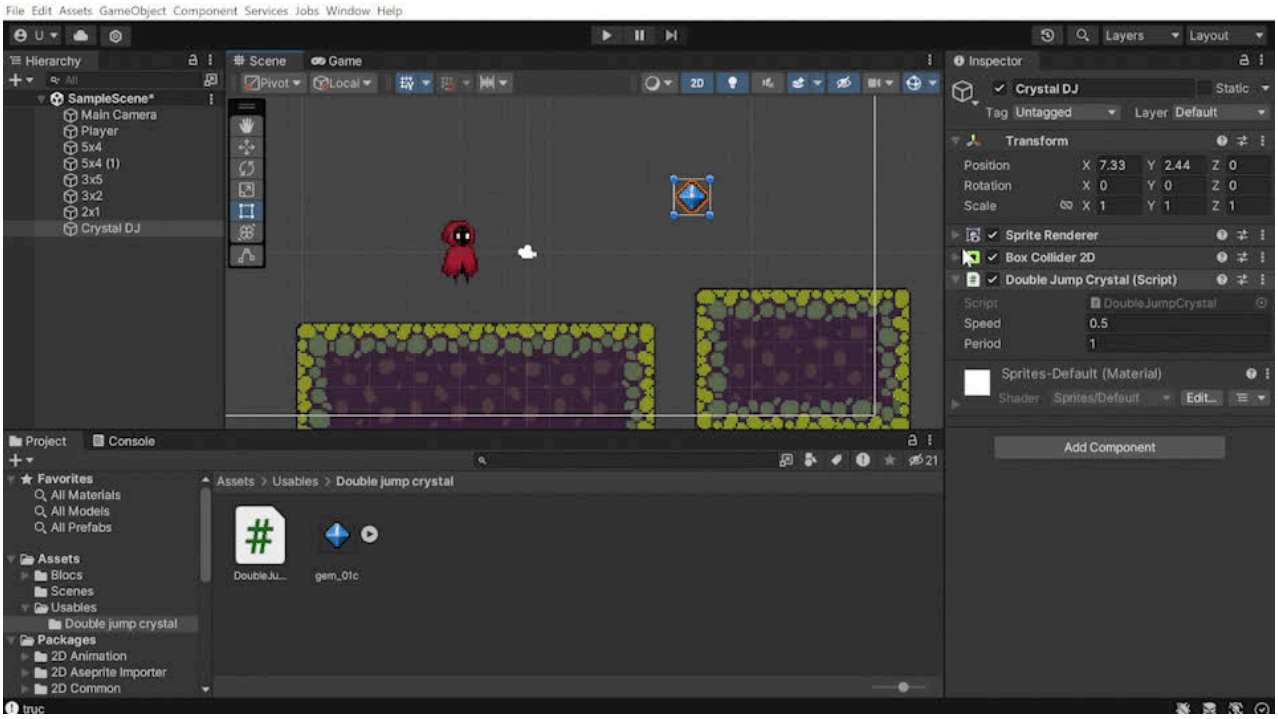
```
public float speed;
public float period;

Message Unity | 0 références
public void Update()
{
    if (Time.time % period > period / 2)
    {
        transform.position += Vector3.up * Time.deltaTime * speed;
    }
    else
    {
        transform.position -= Vector3.up * Time.deltaTime * speed;
    }
}
```



Imaginons maintenant que nous voulons que le cristal disparaisse quand le joueur le touche avant de réapparaitre quelques seconde plus tard.

Pour ce faire il faudrait que le BoxCollider2D et le SpriteRenderer soient tous deux désactivés pendant quelques secondes. Il est possible de désactiver des composants précis dans l'inspecteur. Les composants désactivés n'agiront pas et les fonctions qu'il assurent ne seront pas accomplies.

Par exemple ici en désactivant le SpriteRendere la cristal n'est plus visible mais ses autres composants restent actifs.



Il faudrait que nous puissions faire ce que nous venons de faire dans l'inspecteur mais de manière automatique via le script.

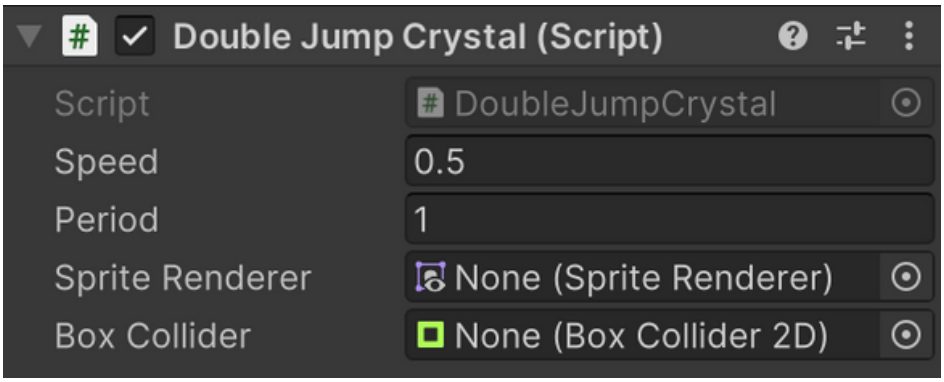
Pour cela nous allons devoir accéder via le script aux composants SpriteRenderer et BoxCollider2D, pour ce faire nous allons créer dans le script du cristal des variables des types des composants. Nous créons en plus une variable de type Float, nous verrons après à quoi elle va nous servir.

```
public class DoubleJumpCrystal : MonoBehaviour
{
    public float speed;
    public float period;

    public SpriteRenderer spriteRenderer;
    public BoxCollider2D boxCollider;
    float timeSinceCollision;
```

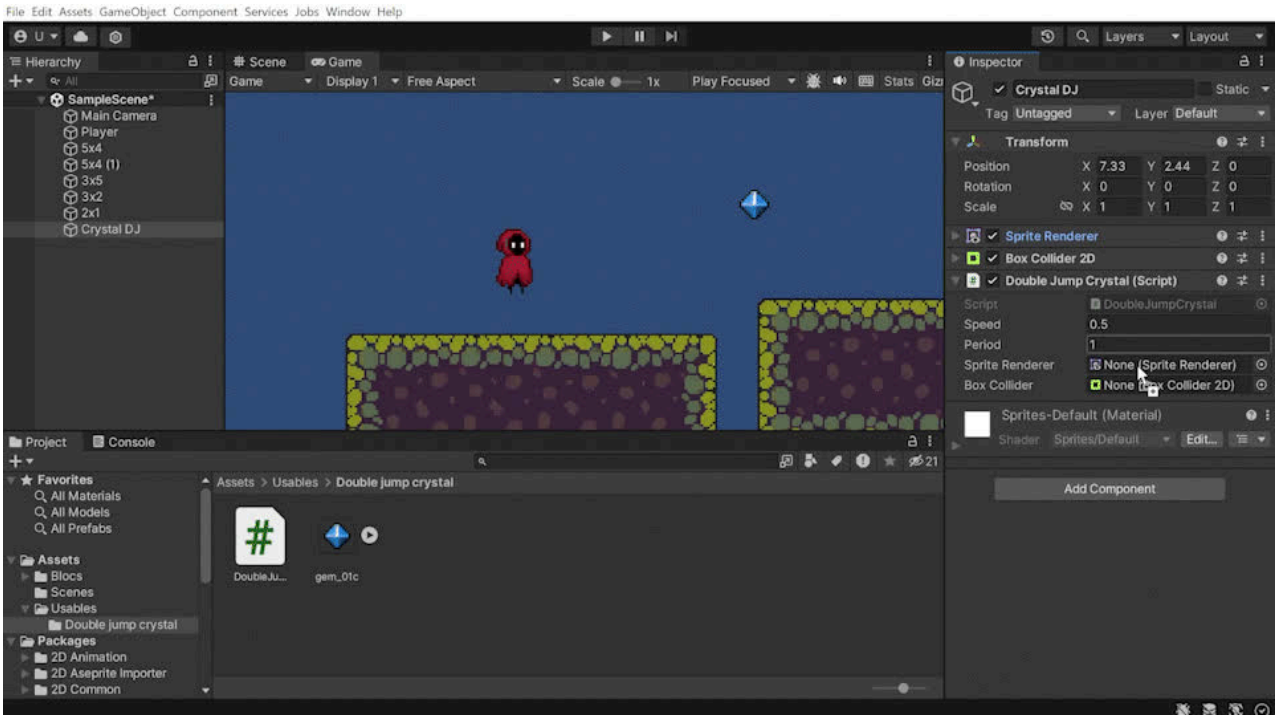
Il faut maintenant ajouter la fonction **OnCollisionEnter2D()** et lui ajouter des instructions pour désactiver les composants quand elle est appelée.

```
private void OnTriggerEnter2D(Collider2D collision)
{
    spriteRenderer.enabled = false;
    boxCollider.enabled = false;
}
```

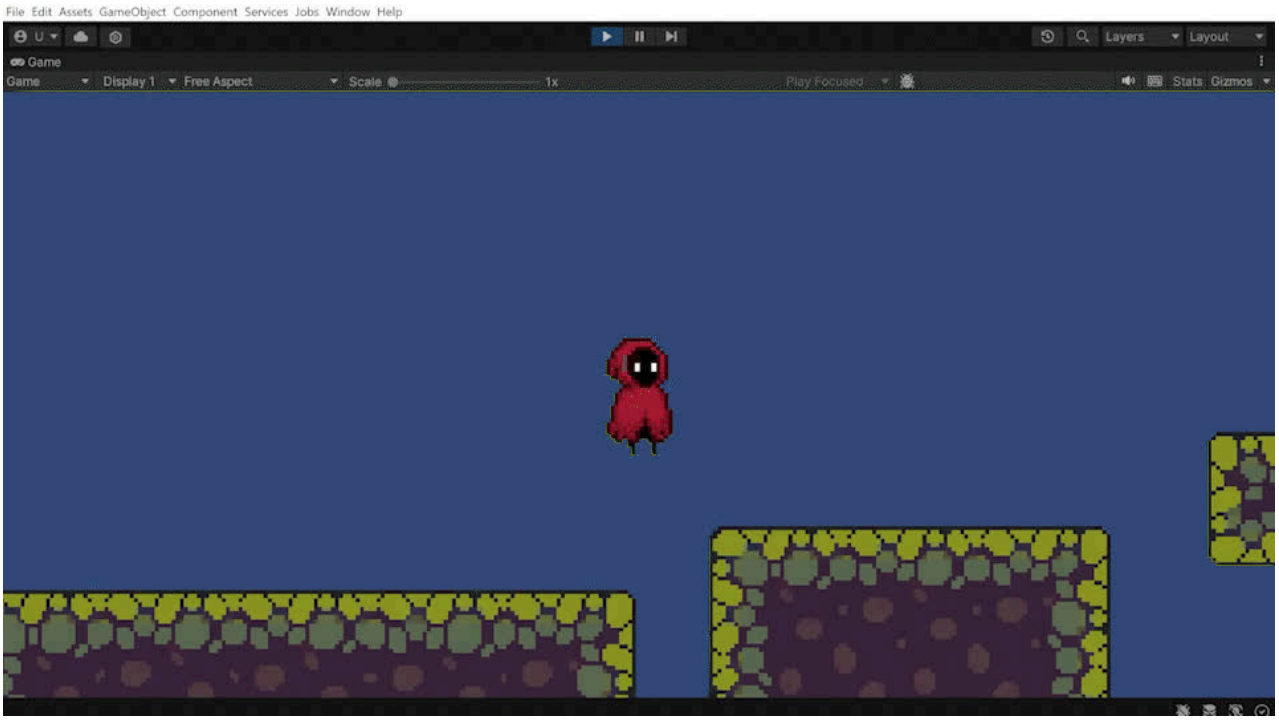
Ici la propriété **enabled** des composants définie si ceux-ci sont activés ou non. Si elle est **False** alors le composant à laquelle elle est liée est désactivé.



Nous voyons que les variables sont bien visibles dans l'inspecteur, il ne nous reste plus qu'à assigner les bons composants.



Si nous testons rapidement nous pouvons voir que le cristal disparait bien mais il ne réapparait pas après. Ceci est logique nous n'avons pas encore mis d'instruction pour qu'il soit réactivé.



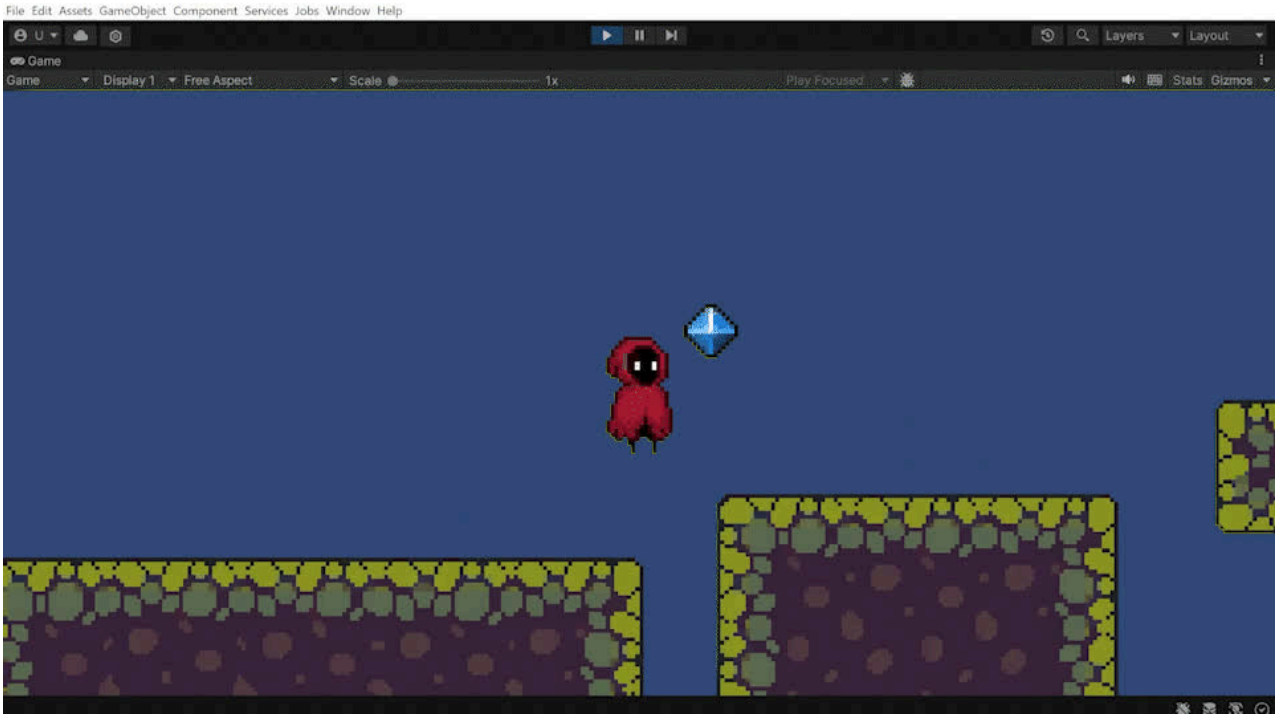
C'est ici que la variable **timeSinceCollision** va entrer en jeu, nous allons l'utiliser

A chaque fois que le cristal sera touché la variable **timeSinceCollision** sera réinitialisée à 0.

```
private void OnTriggerEnter2D(Collider2D collision)
{
    spriteRenderer.enabled = false;
    boxCollider.enabled = false;
    timeSinceCollision = 0;
}
```

Et nous allons rajouter dans **update()** un **if** qui vérifiera si la variable est plus grande que 1.5, si c'est la cas, il réactivera les deux composants, sinon il incrémentera la variable d'un nombre correspondant au temps écoulé depuis la dernière frame (en secondes).

```
if (timeSinceCollision > 1.5)
{
    spriteRenderer.enabled = true;
    boxCollider.enabled = true;
}
else
{
    timeSinceCollision += Time.deltaTime;
}
```



Pour bien comprendre ce qui se passe, regardons attentivement le script.

```
Message Unity | 0 références
public void Update()
{
    if (Time.time % period > period / 2)
    {
        transform.position += Vector3.up * Time.deltaTime * speed;
    }
    else
    {
        transform.position -= Vector3.up * Time.deltaTime * speed;
    }

    if (timeSinceCollision > 1.5)
    {
        spriteRenderer.enabled = true;
        boxCollider.enabled = true;
    }
    else
    {
        timeSinceCollision += Time.deltaTime;
    }
}

Message Unity | 0 références
private void OnTriggerEnter2D(Collider2D collision)
{
    spriteRenderer.enabled = false;
    boxCollider.enabled = false;
    timeSinceCollision = 0;
}
```

2) A chaque frame l'on vérifie si la variable est plus grande que 1.5, si oui on réactive les composants, si non on incrémente la variable du temps qui s'est écoulé depuis la dernière frame

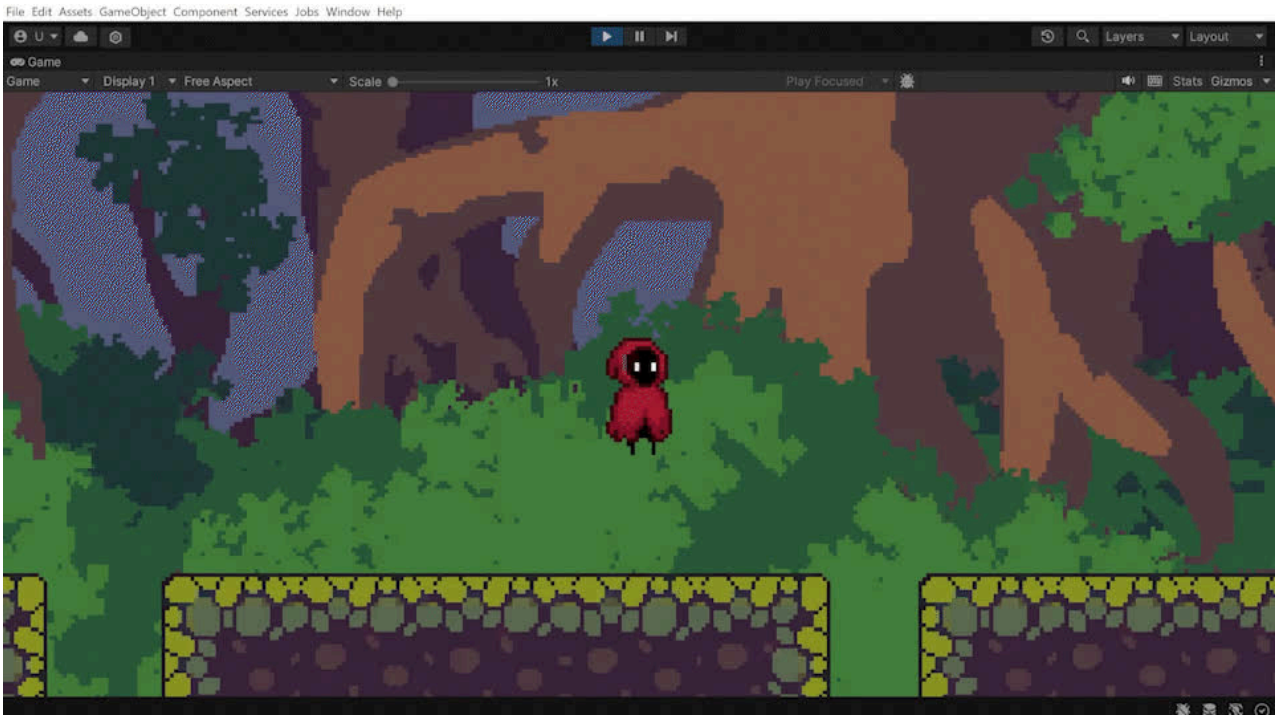
1) Quand le cristal est touché, le sprite et le collider sont désactivés et la variable timeSinceCollision est mise à 0

FOND DÉFILANT

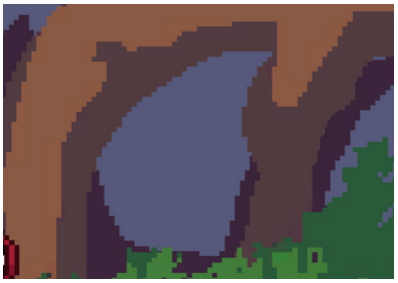
Objectifs :
- Créer un fond défilant quand le/la joueur·euse se déplace

Déroulement de l'activité

Introduction	• Découverte de l'interface collectivement. • Rechercher les différentes zones de travail, ...
Exploration	• Former des binômes, distribuer les fiches et les lancer sur le défi 1. • Les élèves explorent librement l'environnement. Ils n'ont pas de consignes précises mais les inviter à observer les différentes parties de l'interface de programmation.



Dans cet extrait le fond a un bruit dans l'image qui est dû au format GIF dans ce document. Dans le jeu le fond est bien d'un mauve uni.



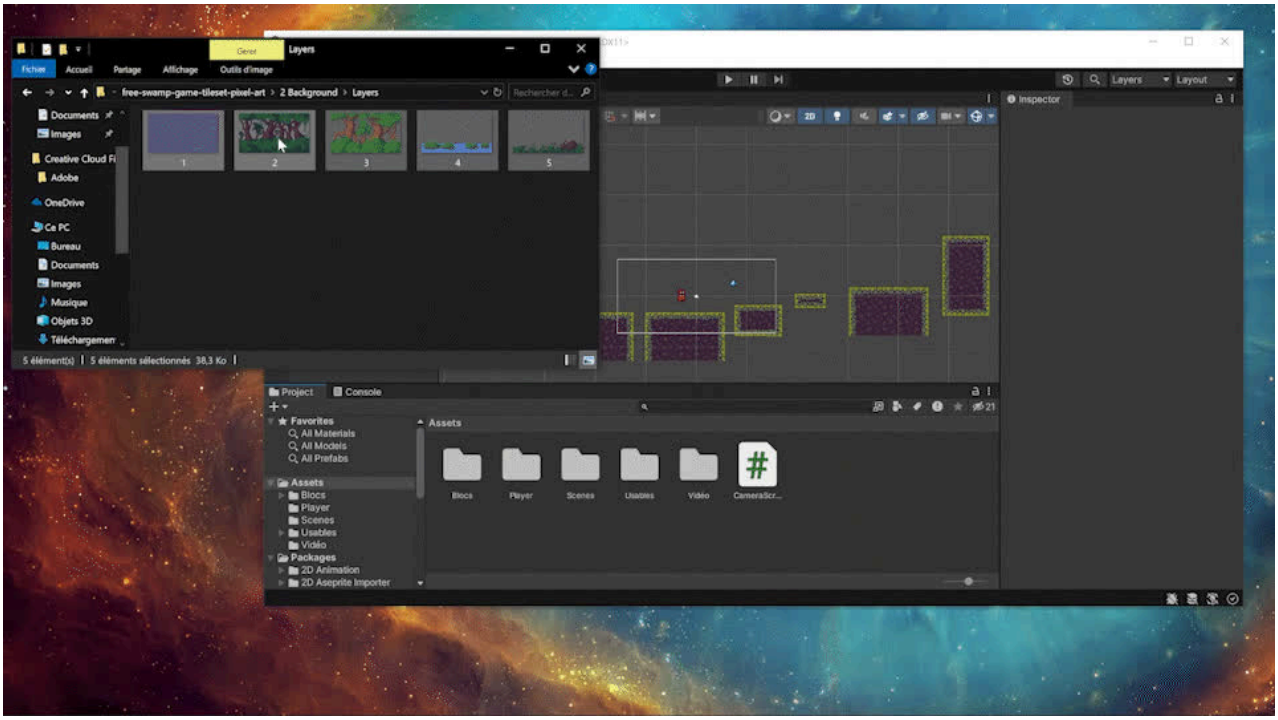
Tout d'abord il faut comprendre ce qui est nécessaire pour avoir un fond (background) défilant. Il va nous falloir plusieurs choses pour arriver à un résultat convaincant :

- Il nous faut une ou plusieurs images de fond qui vont pouvoir se répéter indéfiniment
- Il va falloir faire bouger cette (ces) image(s) en fonction de la position de la caméra
- De plus si l'on utilise plusieurs images pour faire un effet 3D il va falloir déplacer ces images à des vitesses différentes

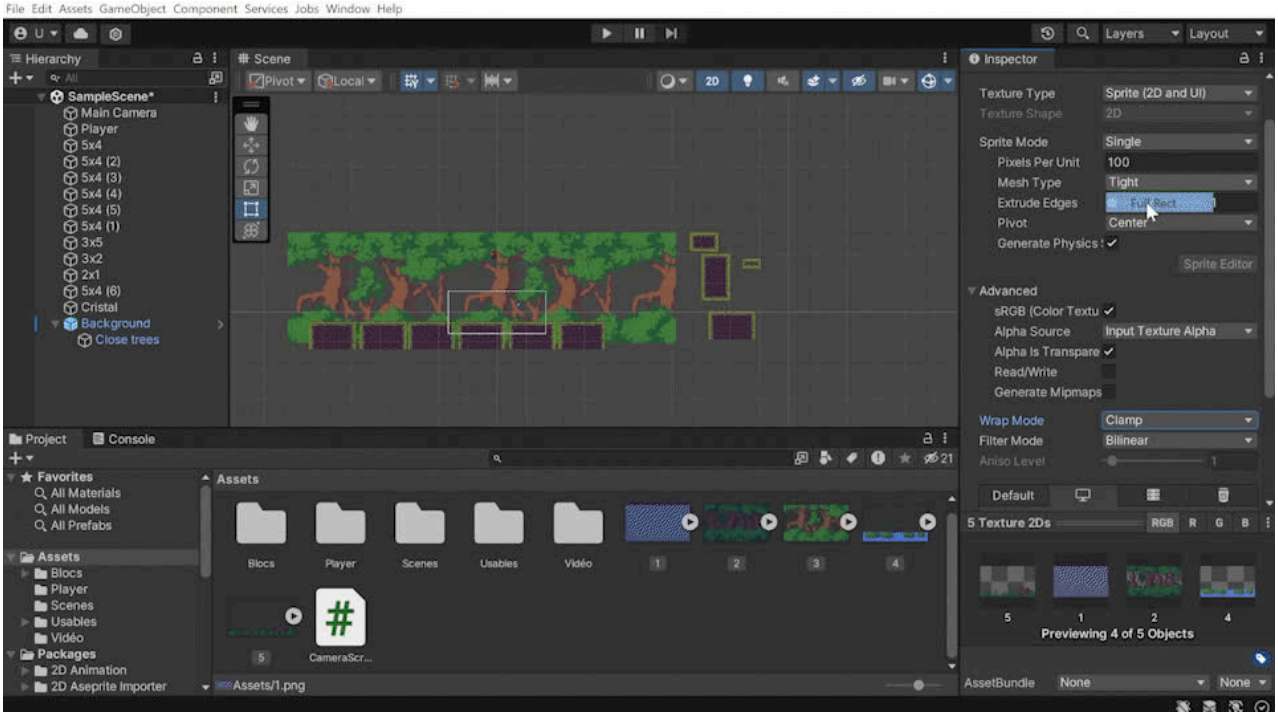
Nous allons donc utiliser des images de fond qui auront été conçues pour présenter un motif qui peut se répéter. Par exemple l'illustration ci-dessous est composée de trois images identiques alignées.

Ce genre d'image peut se trouver sur des sites de partages de ressources.

Il faut donc importer ces images dans Unity.



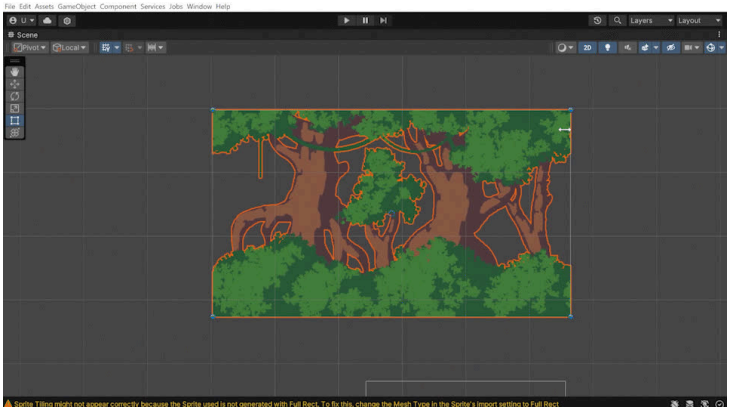
Une fois l'importation effectué il faut correctement paramétrer le sprite.



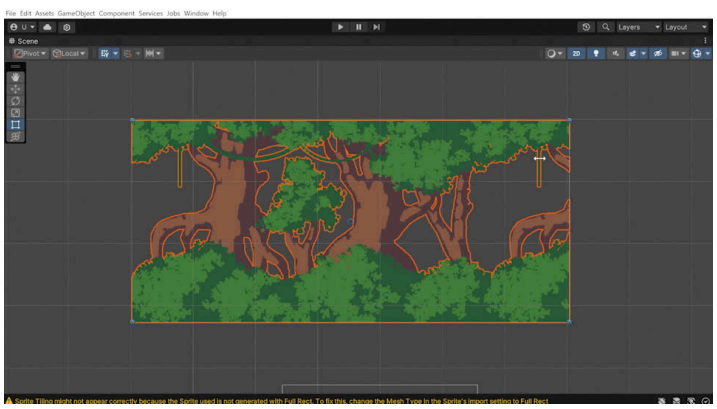
Voyons ensemble les différents paramètres :

- Pixel Per Unit : Nous l'avons vu précédemment, il correspond à l'échelle de taille que prendra le Sprite dans le jeu.
- Filter Mode : Nous l'avons aussi vu précédemment, il permet à des Sprites de faible résolution de ne pas apparaitre floutés.
- Wrap Mode : Il définit la manière dont le sprite sera utilisé avec le tilemode que nous verrons juste après.
- Mesh Type : Il correspond à la manière dont la surface de l'image sera découpé.

Il faut aussi changer un paramètre du SpriteRenderer, le Draw Mode. Il correspond à la manière dont l'image du Sprite sera appliqué sur la zone que l'on désignera pour lui. Avec Clamp le Sprite sera étiré pour occuper toute la surface désignée, tandis qu'avec Repeat il sera répété.

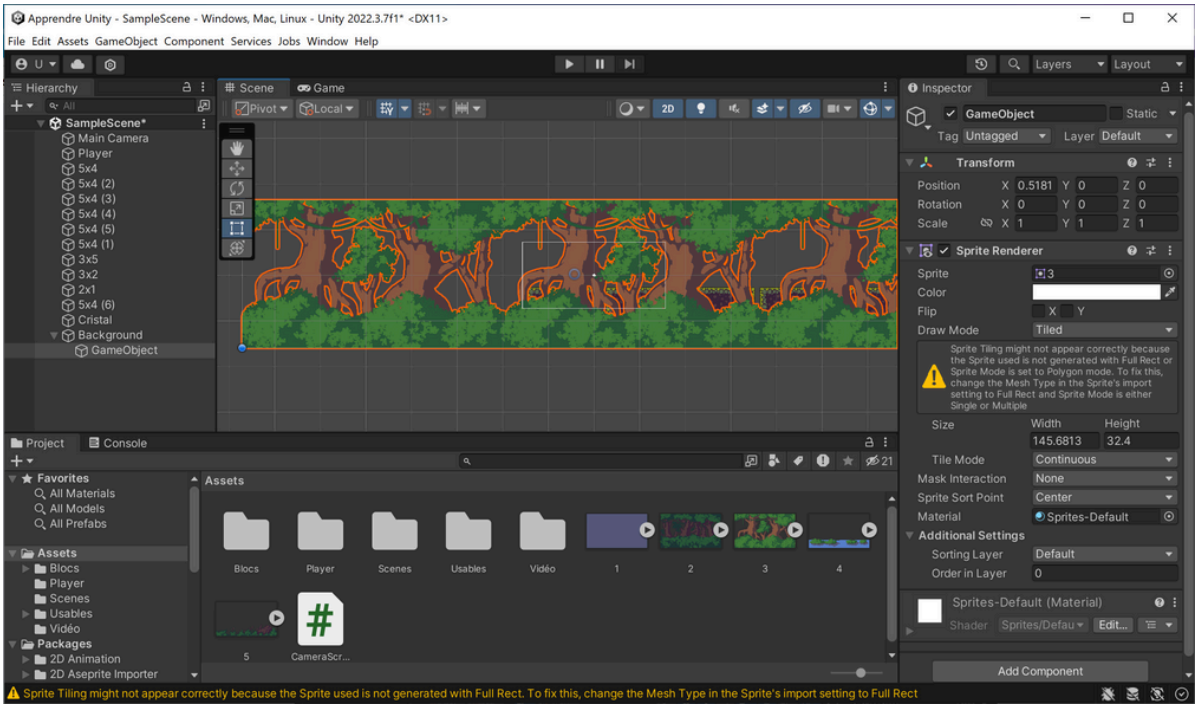


Clamp

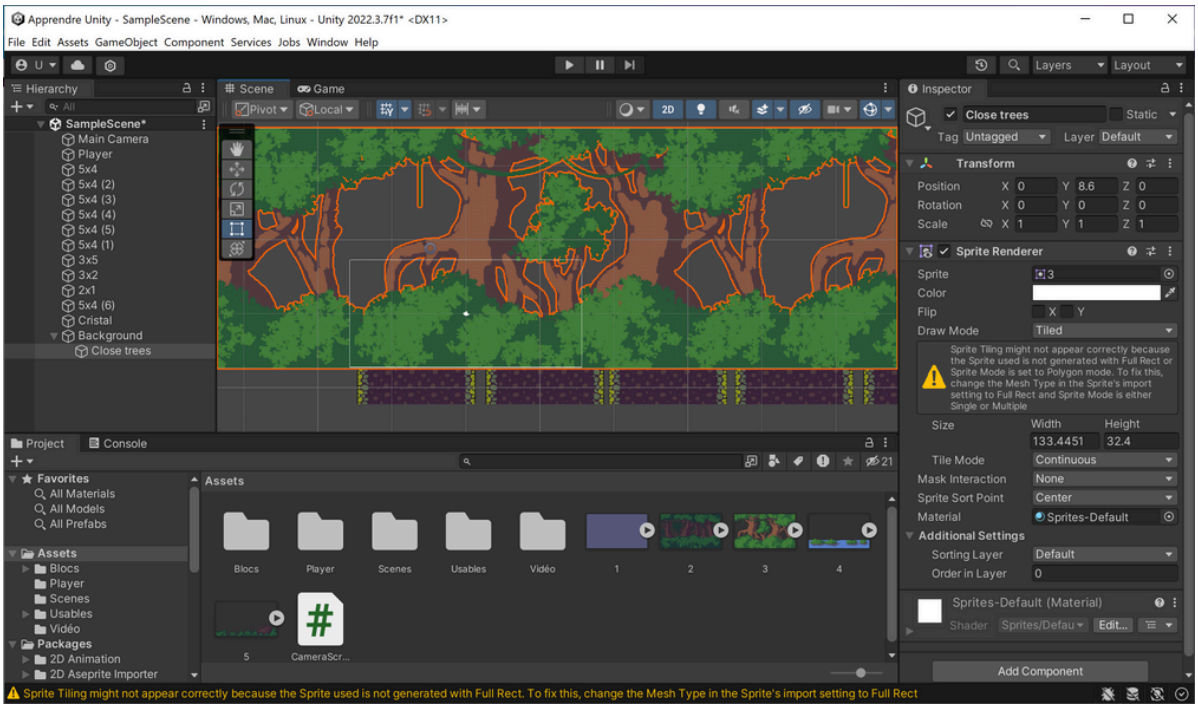


Tiled

Une fois que tout est paramétré, élargissez votre image de manière à ce qu'elle soit au moins trois fois plus large que l'espace de la caméra. Vous pourrez plus tard augmenter cette largeur si nécessaire.



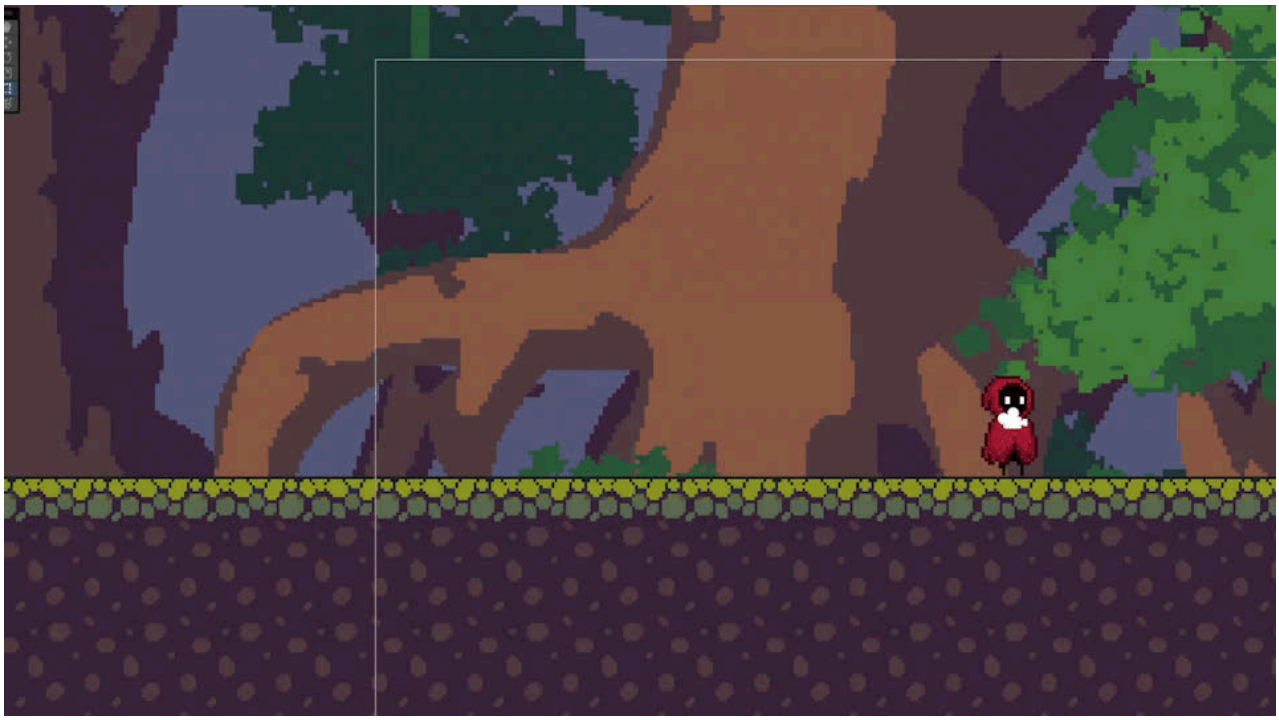
Il est probable que votre image de fond s'affiche au-dessus de celles des éléments de votre niveau.



Pour remédier à ce problème il va vous falloir utiliser les Sorting Layers (voir chapitre éponyme).

Il faut maintenant mettre un script dans le parent qui va le faire bouger avec la caméra de manière à simuler un paysage. Pour ce faire nous allons, à chaque frame, bouger les objets qui contiennent les images.

Ce mouvement devra se faire dans la même direction que la caméra mais à une distance moindre.



L'idée est de comparer la position de la caméra a cette frame-ci et celle qu'elle avait à la frame d'avant. On en fait la différence et on multiplie le résultat par un facteur inférieur à 1. Ceci nous donne le vecteur avec lequel il faut déplacer l'image pour obtenir l'effet de défilement.

```
Script Unity (2 références de ressources) | 0 références
public class BackgroundScript : MonoBehaviour
{
    public float factor;
    public Transform cameraTransform;

    Vector3 lastPos;


    // Start is called before the first frame update
    Message Unity | 0 références
    void Start()
    {
        lastPos = cameraTransform.position;
    }

    // Update is called once per frame
    Message Unity | 0 références
    void Update()
    {
        transform.position += (cameraTransform.position - lastPos) * factor;
        lastPos = cameraTransform.position;
    }
}
```

Une fois que vous avez créé votre objet avec une image de fond et le script associé il ne reste plus qu'à copier cet objet en plusieurs exemplaires.
Vous pourrez mettre une image différentes pour chacun d'entre eux, spécifié l'ordre dans lequel ils s'affichent et leurs facteurs de défilement.

Plus le facteur est proche de 1 plus l'objet bougera lentement par rapport à la caméra et donc plus l'objet paraitra loin.

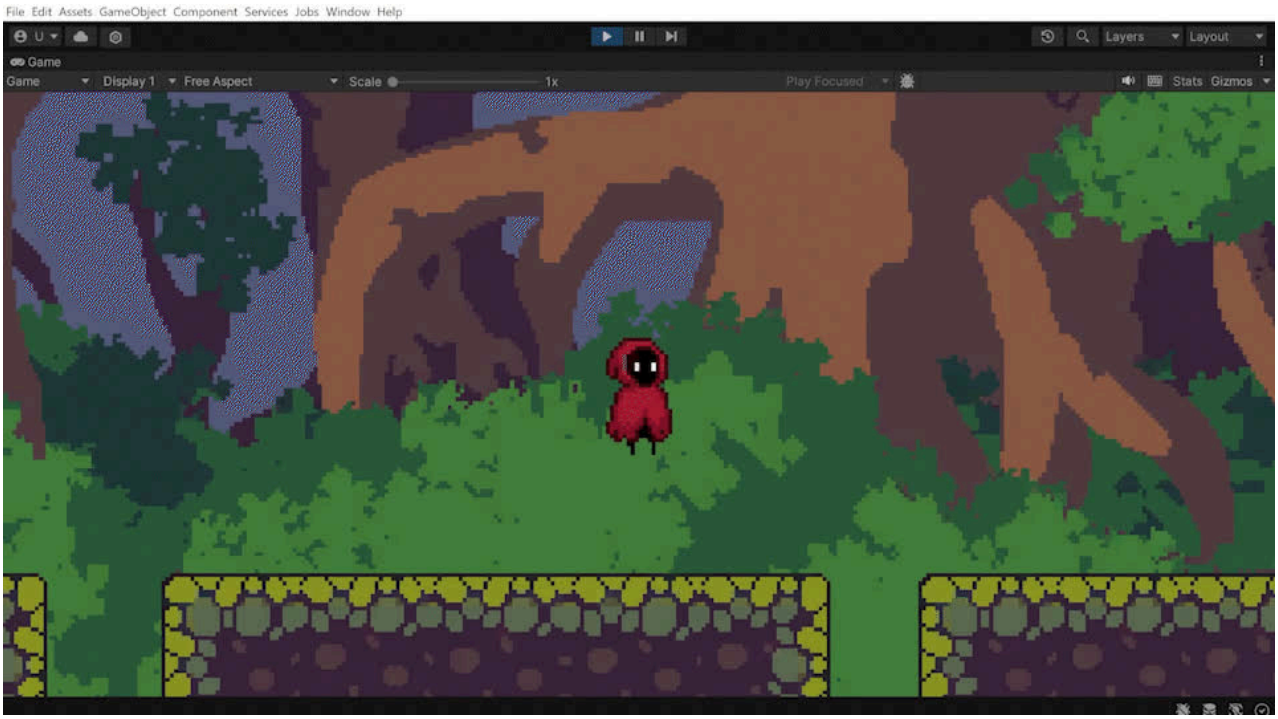
Donc la règle à suivre est : plus un fond est sensé être loin plus son facteur est proche de 1 et plus son Ordre dans la couche (Order In Layer) est petit.

ANIMATION

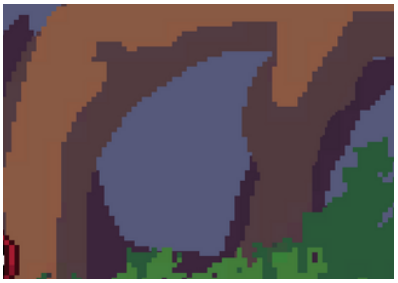
Objectifs :
- Animer des sprites

Déroulement de l'activité

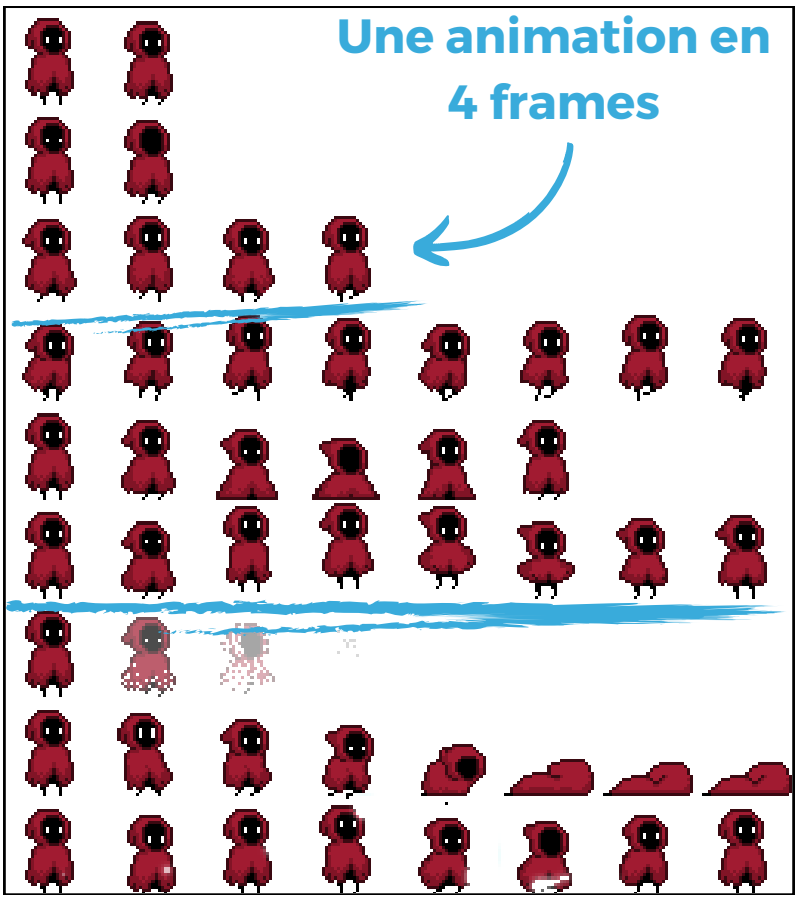
Introduction	• Découverte de l'interface collectivement. • Rechercher les différentes zones de travail, ...
Exploration	• Former des binômes, distribuer les fiches et les lancer sur le défi 1. • Les élèves explorent librement l'environnement. Ils n'ont pas de consignes précises mais les inviter à observer les différentes parties de l'interface de programmation.



Dans cet extrait le fond a un bruit dans l'image qui est dû au format GIF dans ce document. Dans le jeu le fond est bien d'un mauve uni.

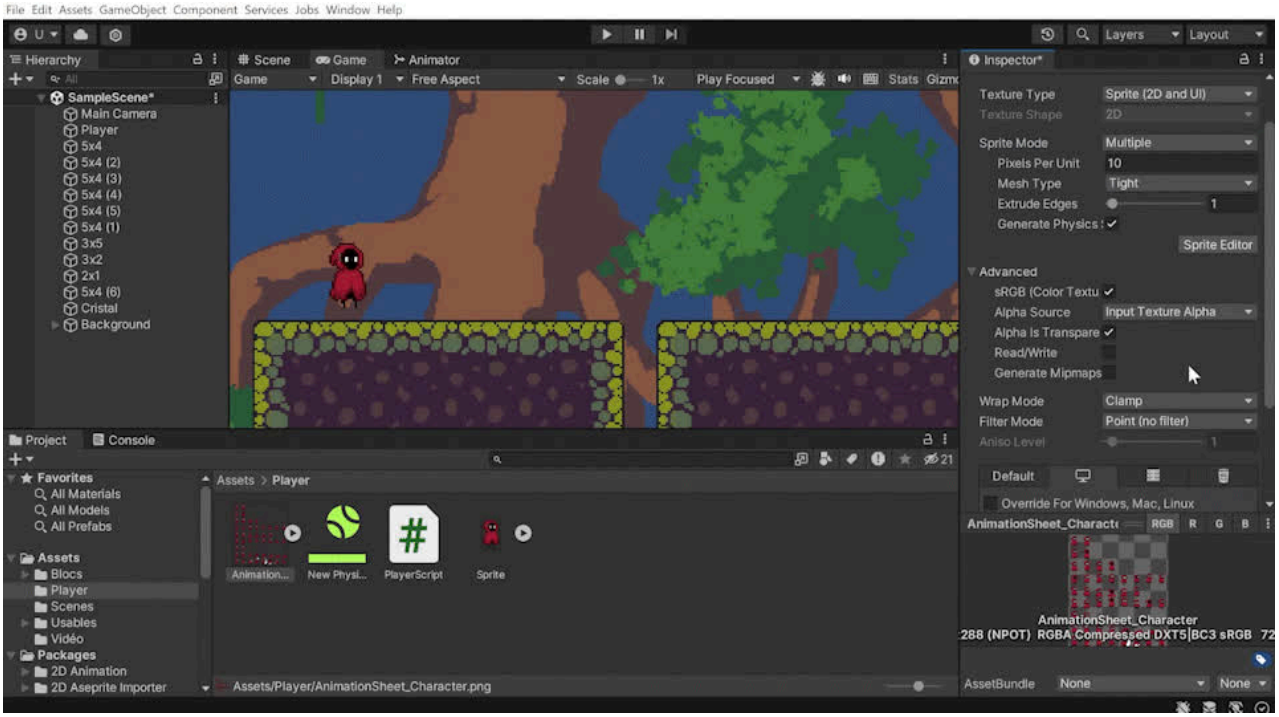


Commencez par importer les images qui composent votre animation. Si vous cherchez des sprites en ligne, vous trouverez régulièrement ce que l'on appelle des spritesheets (feuilles de sprite). Une spritesheet est une image qui va regrouper les différentes frames (prononcé comme ceci) des animations de l'élément.

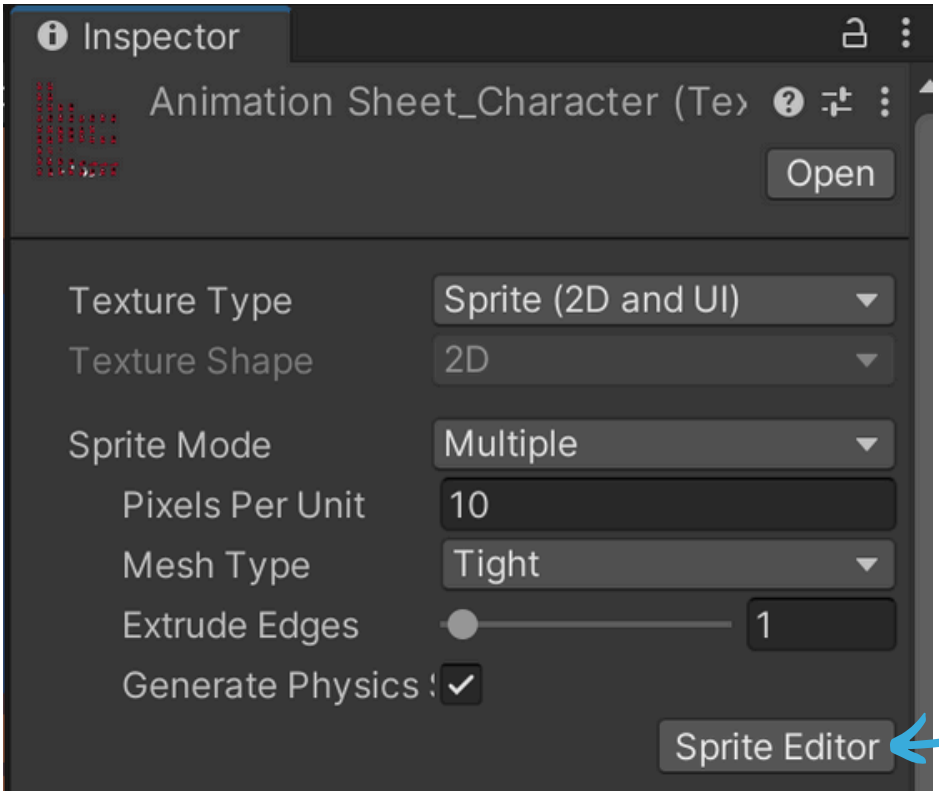


Si vous utilisez une spritesheet, il va falloir commencer par "découper" les frames. Sinon passez directement à "Créer une animation".

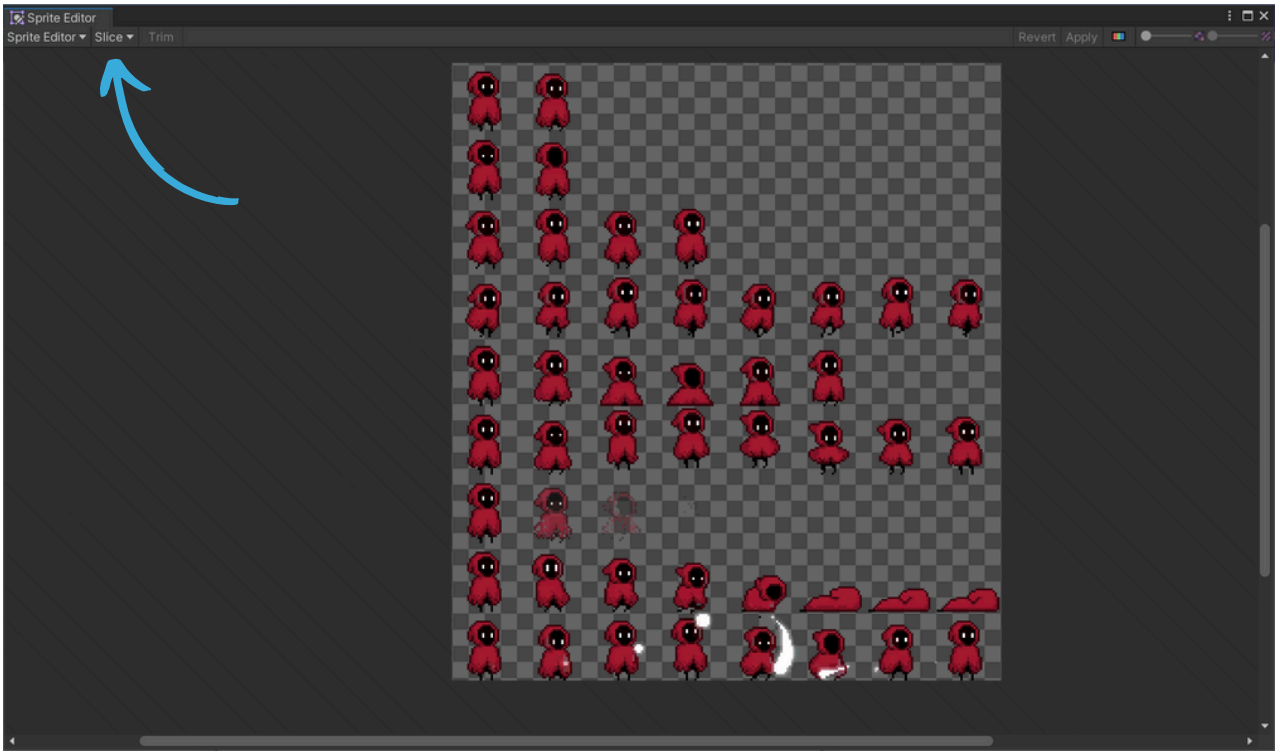
Commencez par sélectionner l'image.
Allez dans ses propriétés.
Changez les Sprite Mode en le mettant à "Multiple".
Cliquez sur "Apply".



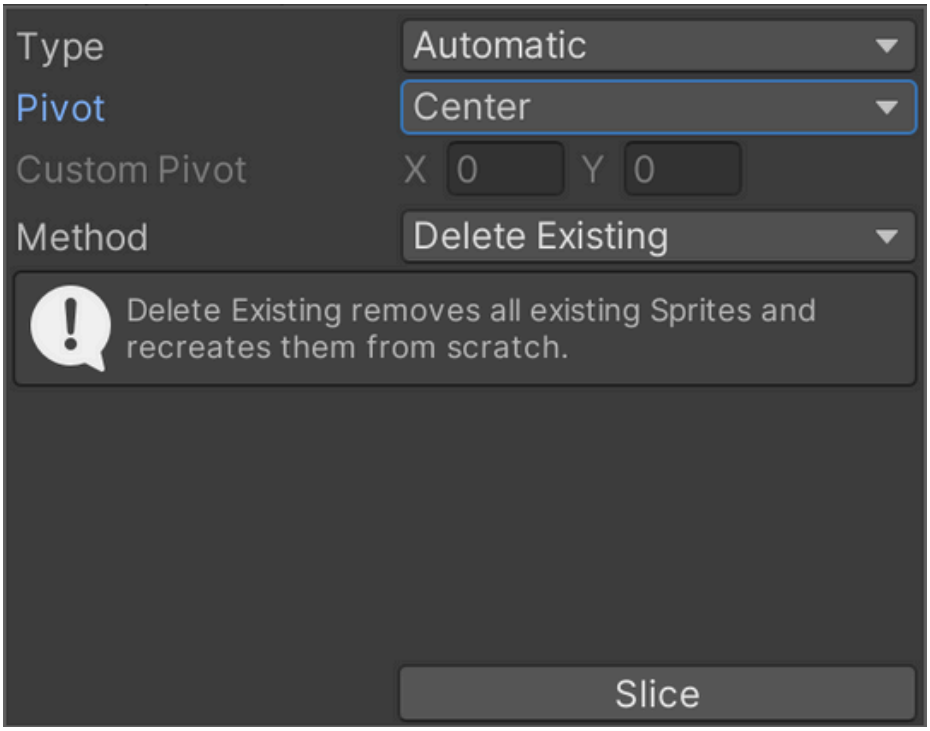
Toujours dans les propriétés de votre image, cliquez ensuite sur "Sprite Editor".



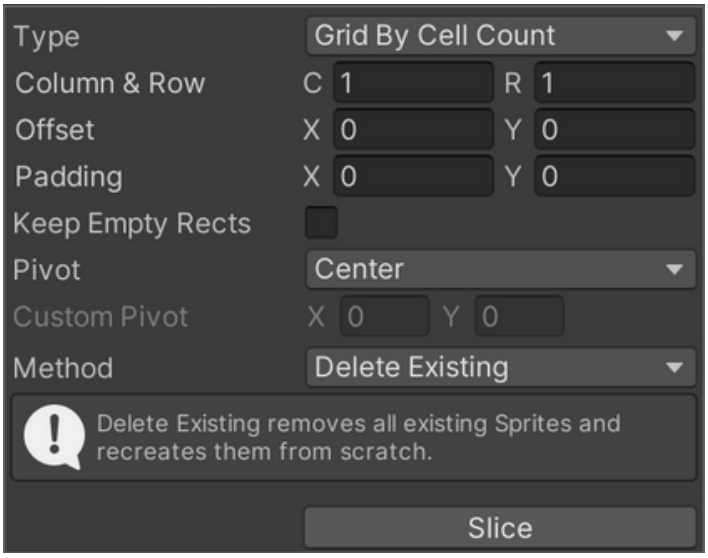
Une fenêtre va s'ouvrir, elle sert à découper les différentes frames dans l'image.
Cliquez sur "Slice" (découper).



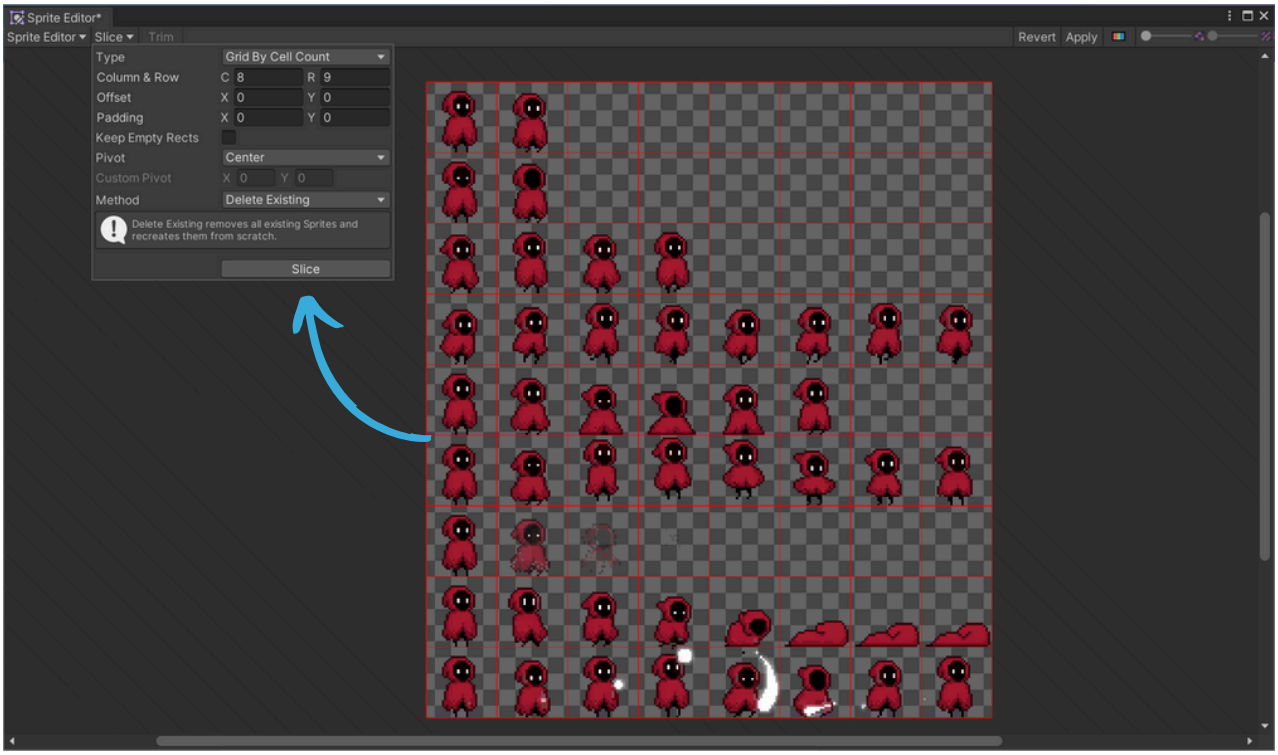
Une petite fenêtre va s'ouvrir avec des paramètres de découpe.
Nous allons utiliser le type de découpe "Grid by cells count".



La fenêtre va changer, de nouveaux paramètres vont apparaitre. Nous allons changer les “Column & Row”, ils correspondent au nombre de colonnes et de lignes.

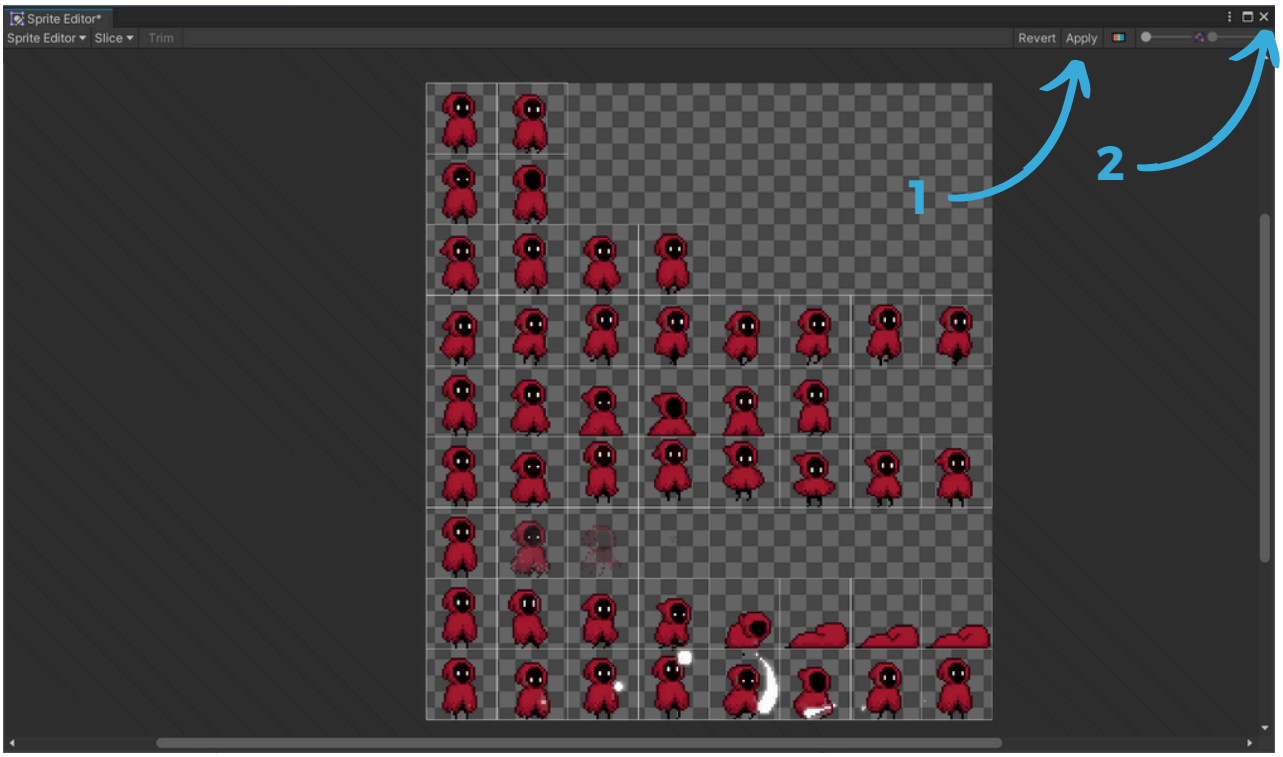


La spritesheet utilisée plus haut compte 9 lignes et 8 colonnes, ce sont les paramètres que je vais utiliser.



Une fois les paramètres entrés des lignes rouges s’affichent. Elles vous indiquent le découpage que vous avez paramétré. S’il correspond bien à ce à quoi vous vous attendiez, cliquez sur le “Slice” de cette nouvelle fenêtre.
Cette dernière étape effectuera la découpe.

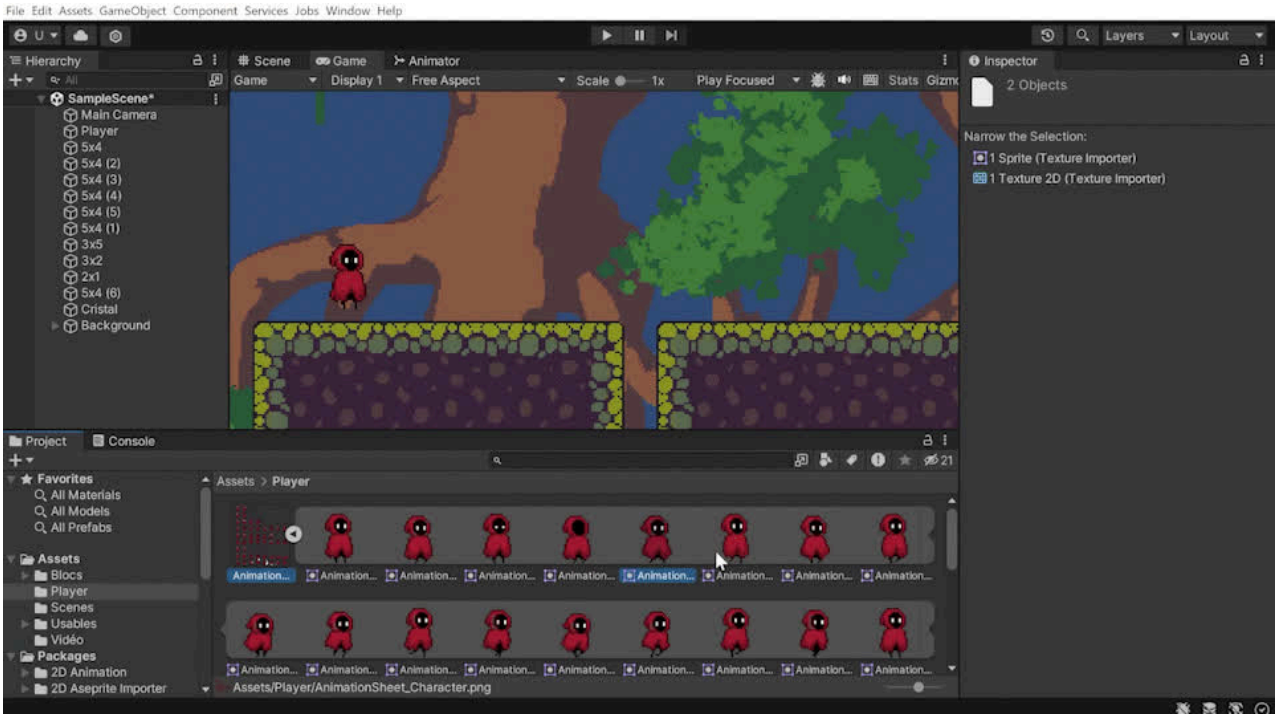
Vous vous retrouverez avec les différentes frames découpées et encadrées avec des traits blancs.



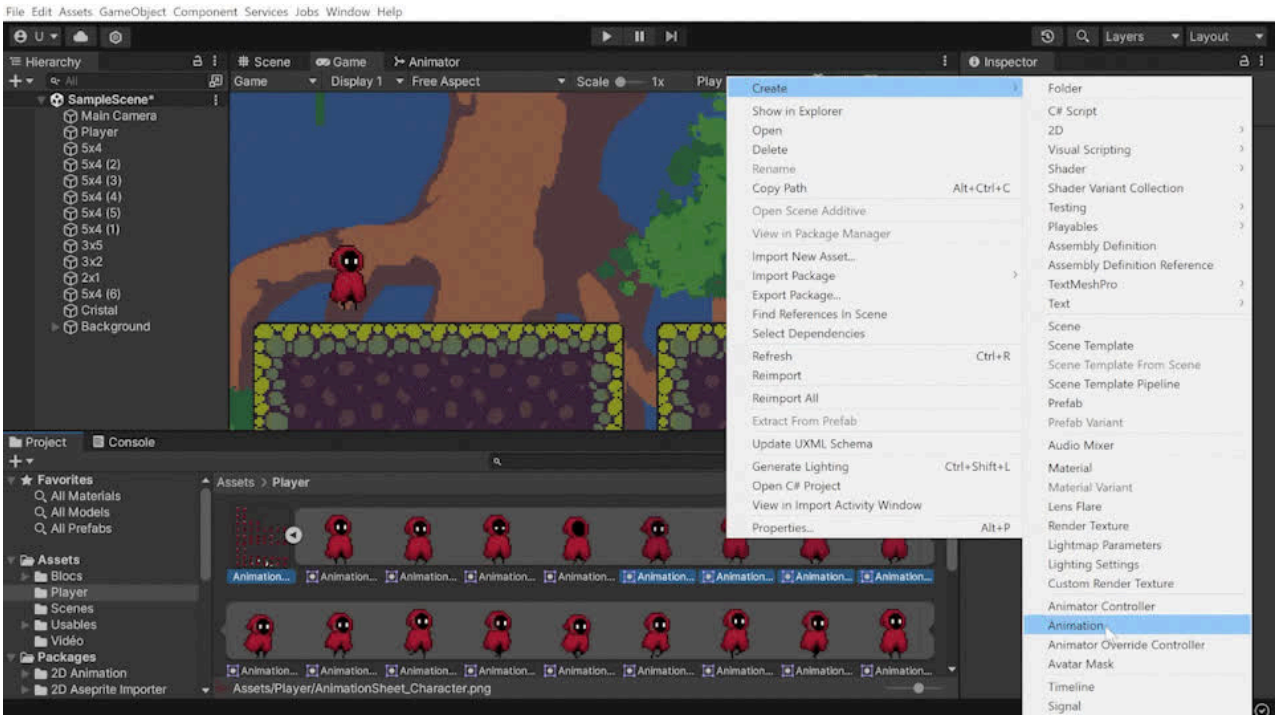
Cliquez finalement sur “Apply” puis sur la croix en haut à droite pour terminer et revenir à l’éditeur Unity.

Créer une animation

Une fois vos sprites créés, il va falloir réaliser votre animation. Pour ce faire, sélectionnez dans le bon ordre les différentes frames de votre animation (maintenez la touche **ctrl** pour sélectionner plusieurs éléments).

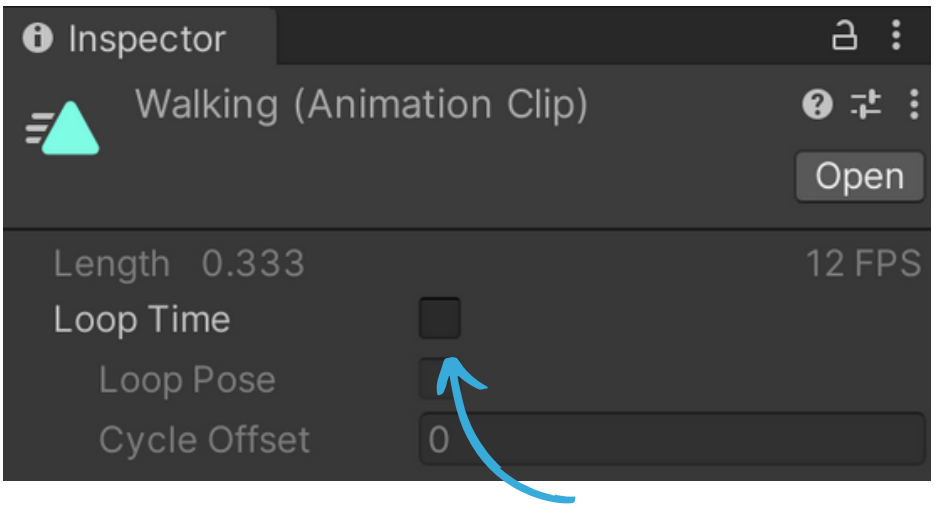


Faites ensuite un clic droit, sélectionnez “Create” puis “Animation”.



Vous verrez alors un élément “Animation” créé et vous pourrez le renommer pour ne pas le perdre.

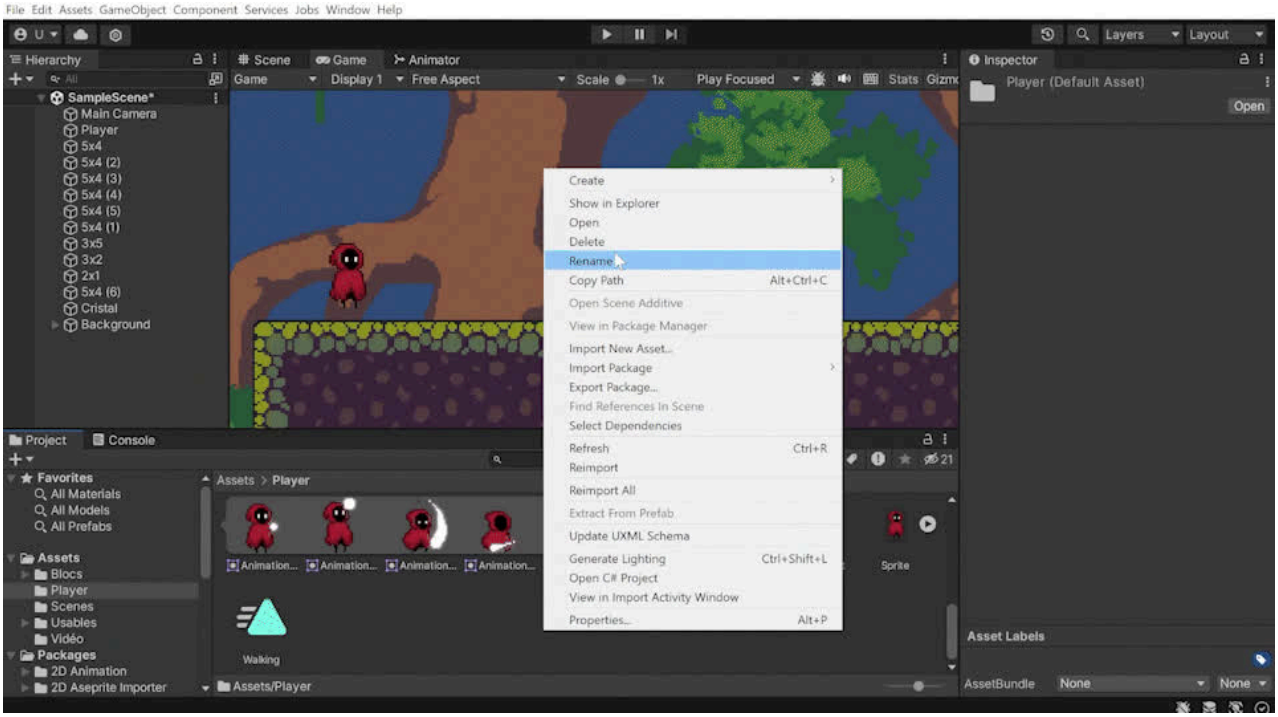
Il ne vous reste plus qu'à sélectionner votre animation et activer le "Loop Time". Ce paramètre définit si l'animation va se répéter à chaque fois qu'elle est finie.



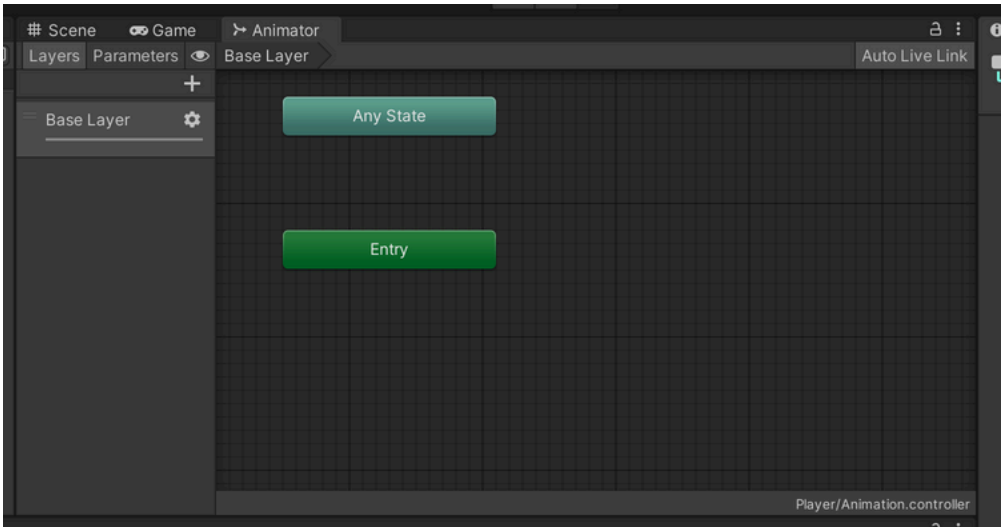
Animer un GameObject

Ajoutez ensuite un composant “Animator” au GameObject que vous voulez animer.

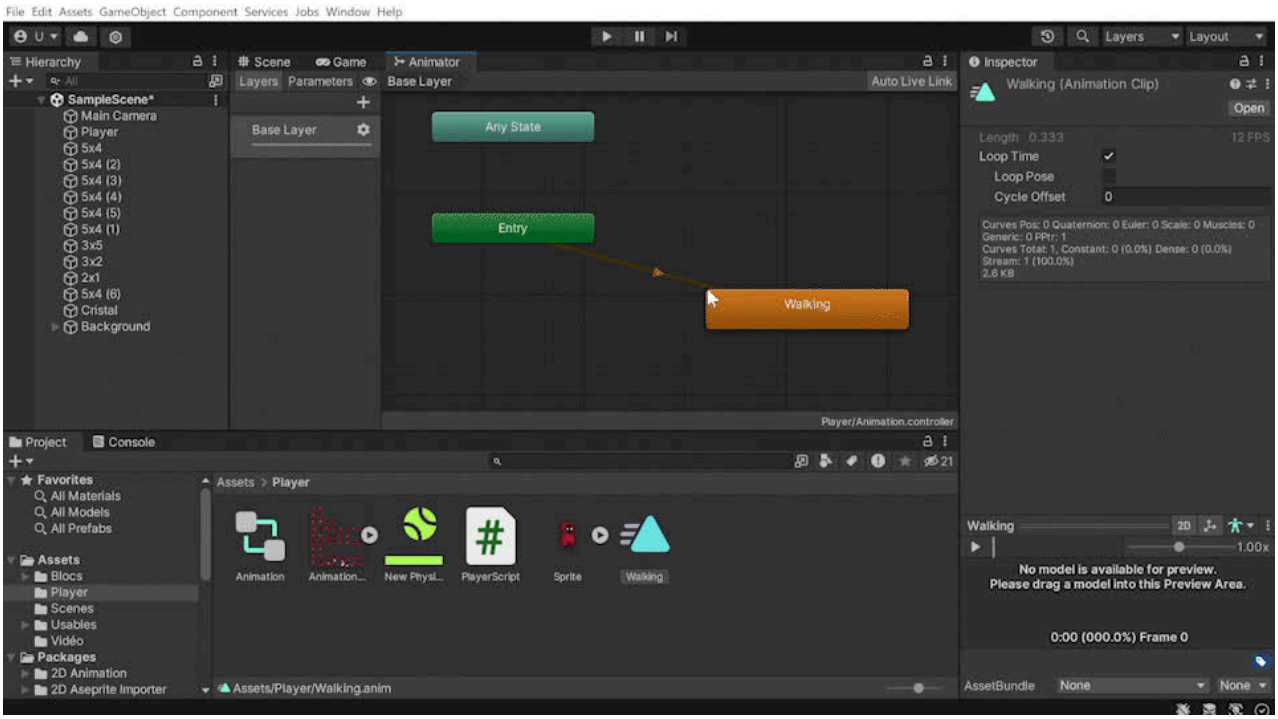
Dans l’explorateur de fichier, faites un clic droit, cliquez sur “Create” puis sur “Animator Controller”.



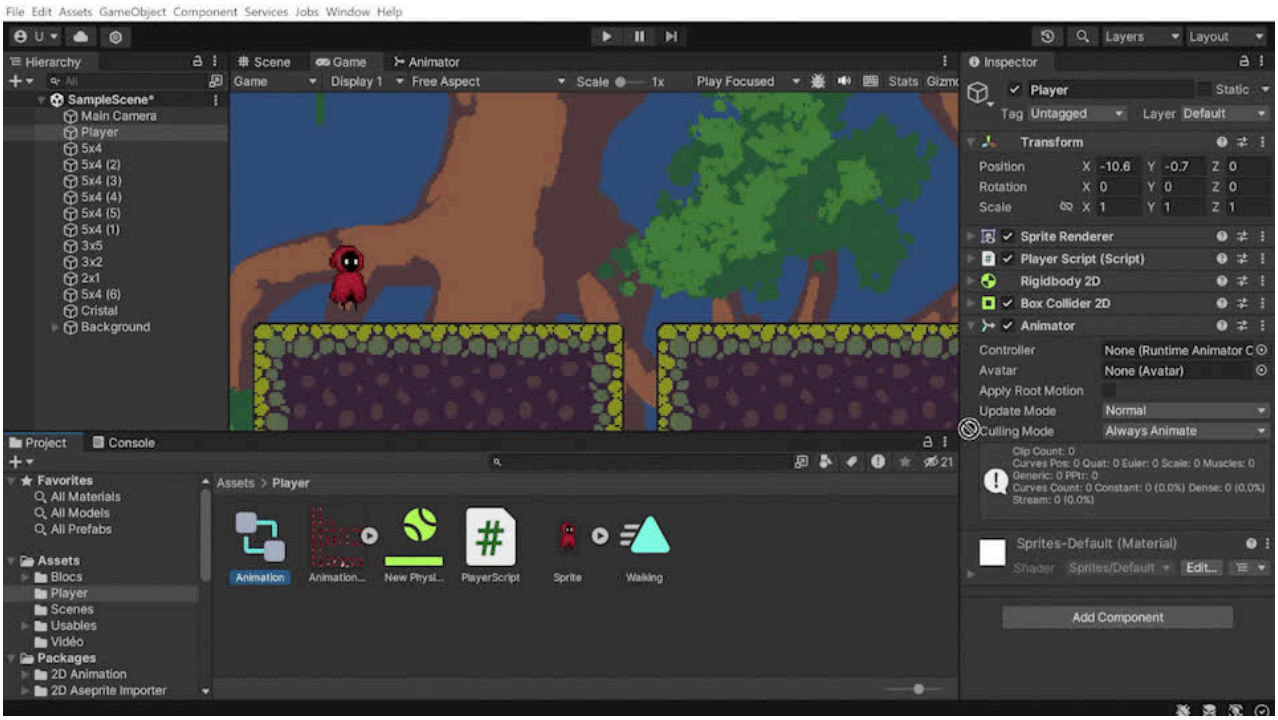
Double-cliquez ensuite sur l’élément que vous venez de créer. Une nouvelle fenêtre s’ouvrira.



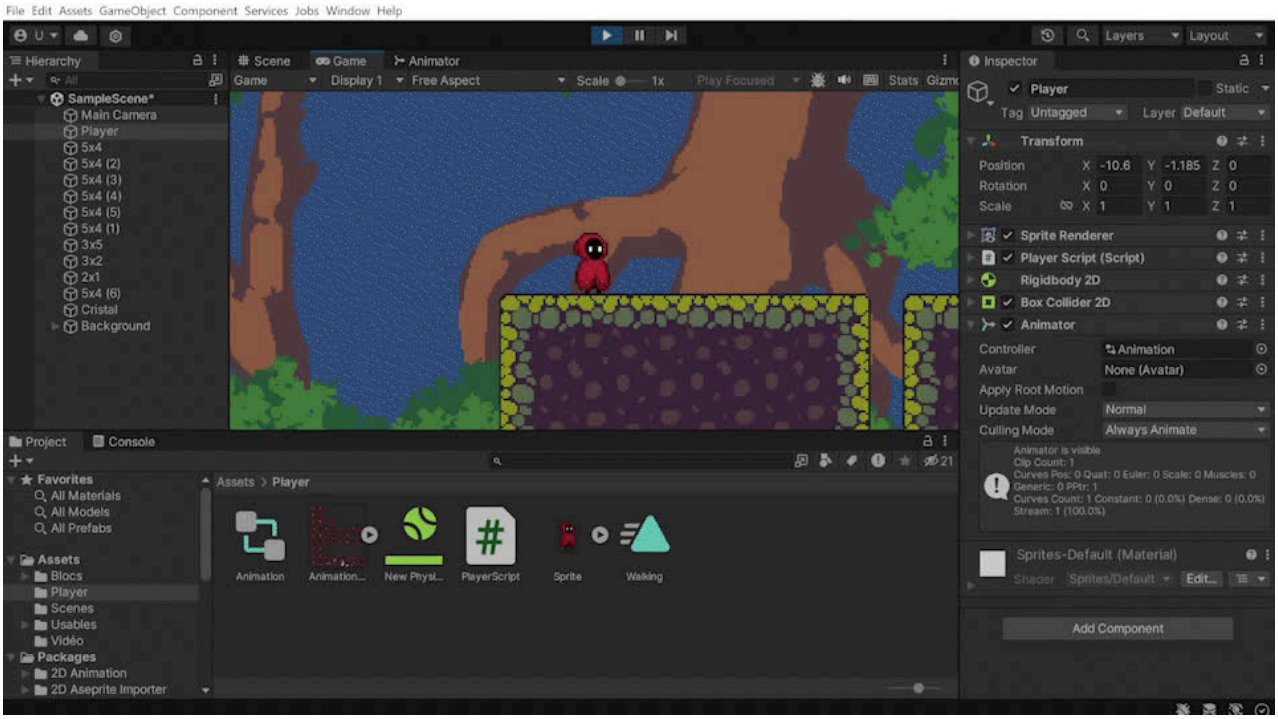
Dans cette fenêtre, glissez-déposez l'animation que vous avez créée.



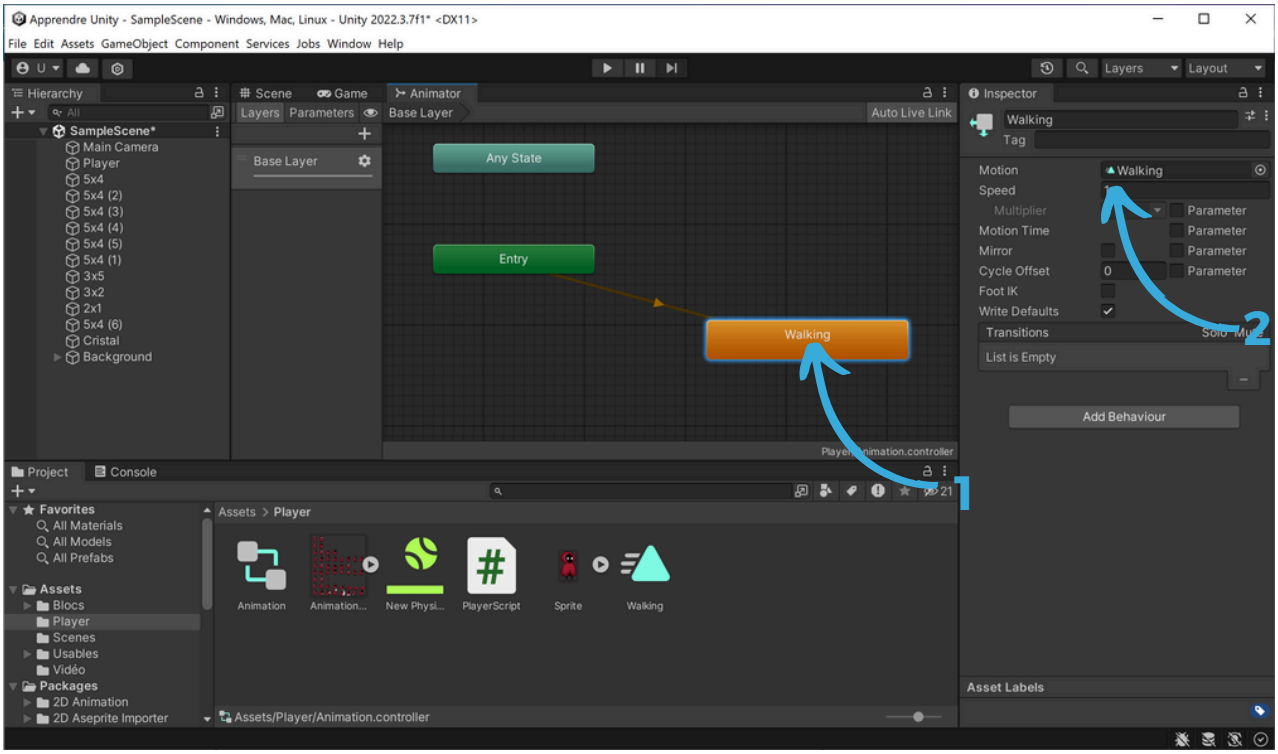
Sélectionnez maintenant le GameObject que vous voulez animer et glissez-déposer l'Animator Controller dans la propriété "Controller" du composant "Animator".



Si nous lançons le jeu, nous pouvons voir que le personnage s'anime bien mais peut-être pas à la bonne vitesse.



Pour régler ce problème, retournez dans la fenêtre d'édition de l'animator.
Sélectionnez le rectangle créé lorsque vous avez déposé de votre animation.



Il ne vous reste plus qu'à changer la valeur de la Speed (vitesse) pour changer la vitesse de l'animation.

ANIMATOR CONTROLLER

Qu'est-ce que c'est

L'Animator Controller (contrôleur d'animateur), comme son nom l'indique, est un composant qui va permettre de contrôler quelle animation est jouée à quelle moment.
Il va être possible de lui donner plusieurs animations et c'est lui qui va s'occuper de les lancer selon ses paramètres.

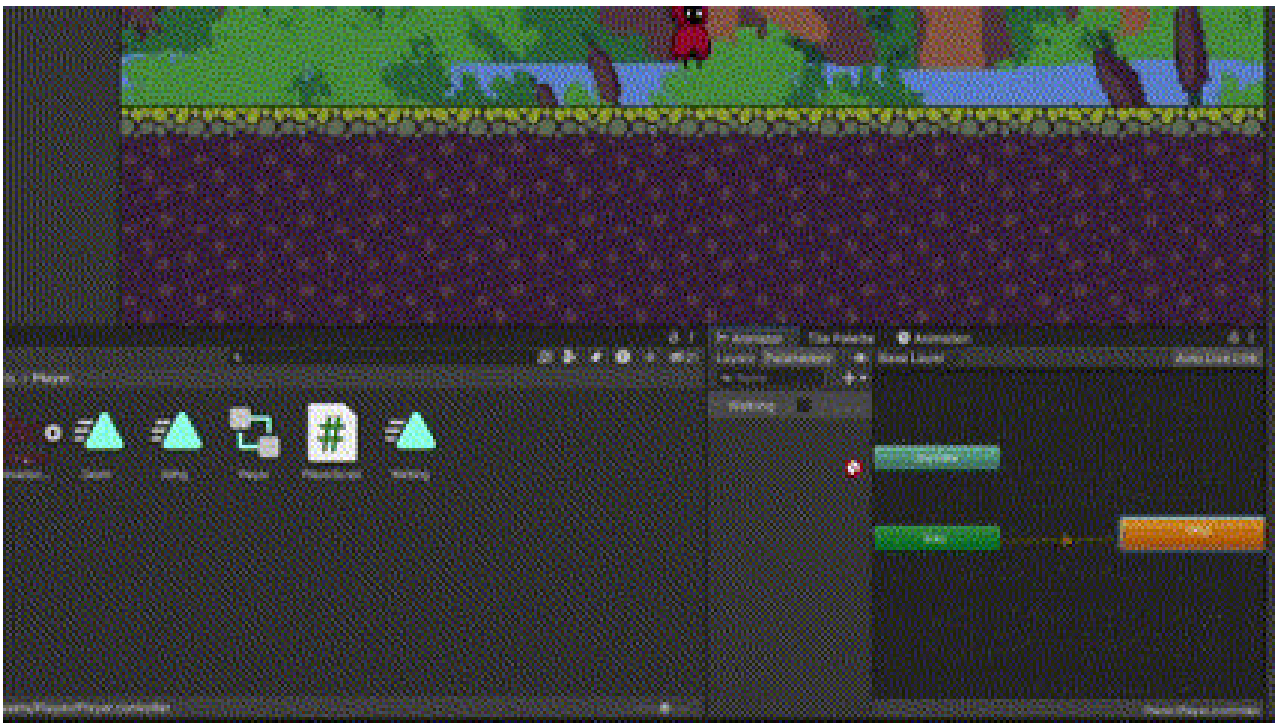
State machine

Une chose importante à comprendre avec l'Animator Controller est que c'est ce qu'on appelle une state machine (machine d'état). C'est à dire un programme qui va avoir un certain nombre d'états possible et qui se comportera différemment en fonction de l'état dans lequel il se trouve.

Dans le cas de notre Animator Controller chaque animation va être un état indépendant. Donc une et une seule animation peut être jouée à la fois.

Passer d'une animation à une autre

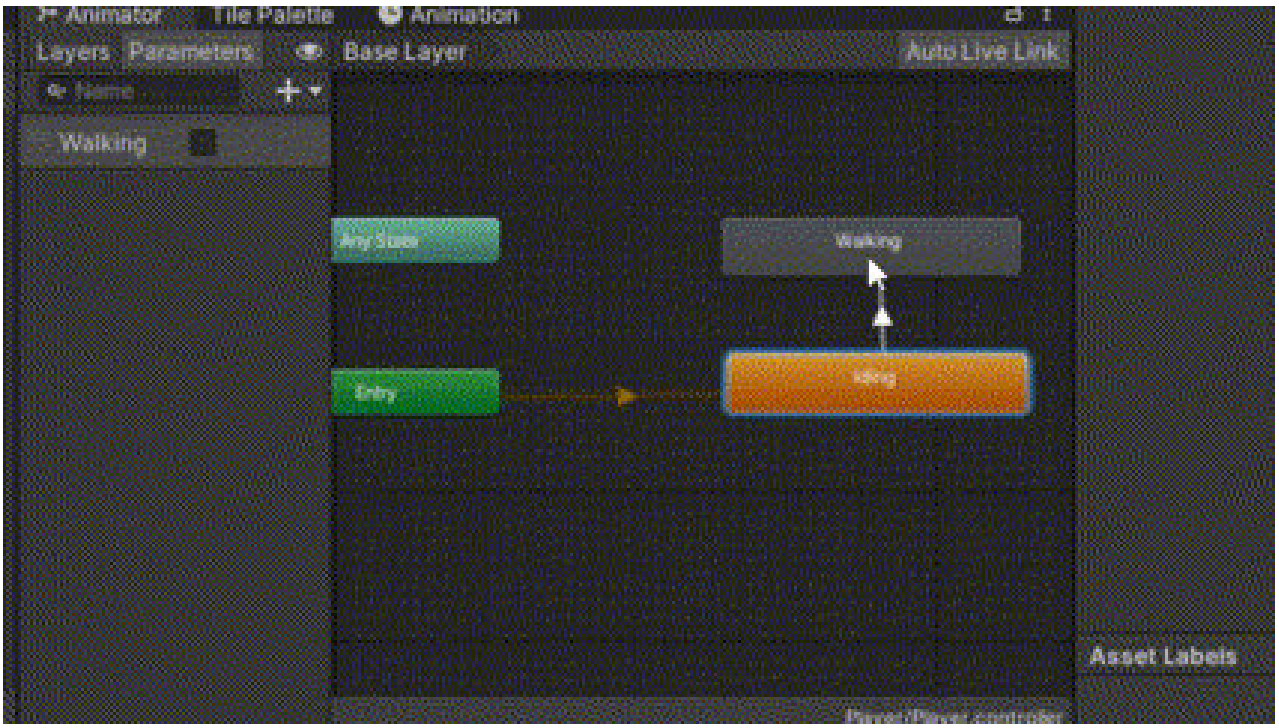
Si nous ajoutons une nouvelle animation à notre Animator Controller nous pouvons voir qu'elle reste grisée et n'a pas de petite flèche qui y mène.
Nous allons voir de suite ce que ça veut dire et comment y remédier.



Transition et Entry
Les flèches entre les animations sont ce qu'on appelle des transitions, ce sont des choses qui permettent de passer d'une animation à une autre. Ces transitions ne se déclenchent pas systématiquement mais nécessitent généralement que certaines conditions soient remplies (nous verrons ça plus loin).

Quand nous avons ajouté une animation nous avons pu voir qu'un transition entre Entry et notre animation a été automatiquement créée. Cette dernières est de couleur orange/jaune. Entry correspond à l'entrée dans la state machine, c'est le point de départ et l'animation qui y est connectée sera jouée par défaut.

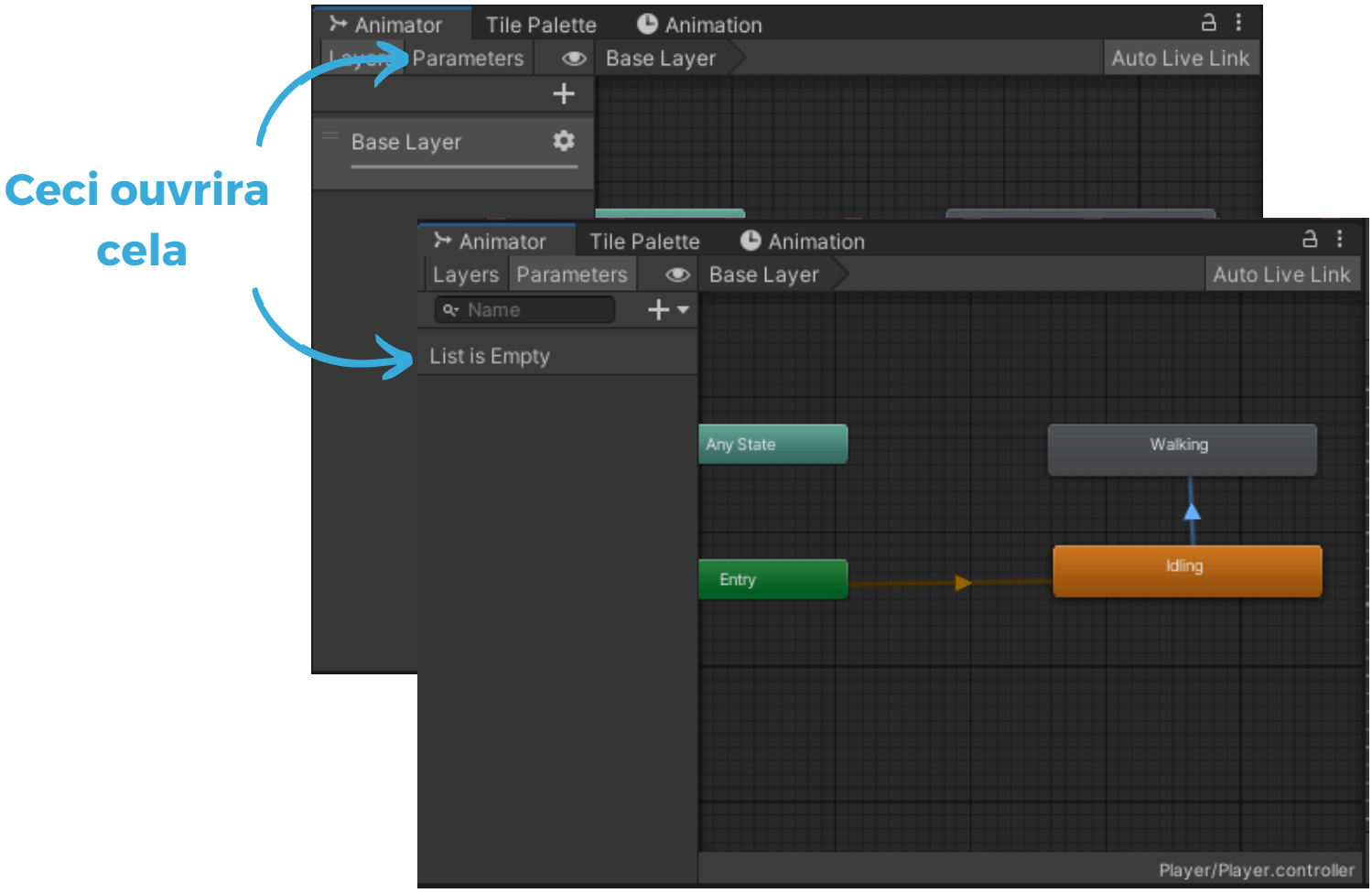
Si nous voulons qu'il soit possible de passer d'une animation à une autre il va falloir créer de nouvelles transitions. Pour cela faite clique droit sur l'animation de départ et sélectionnez "Make Transition" une flèche va suivre votre curseur. Avec la flèche attachée à votre curseur cliqué sur l'animation vers laquelle vous voulez créer une transition.



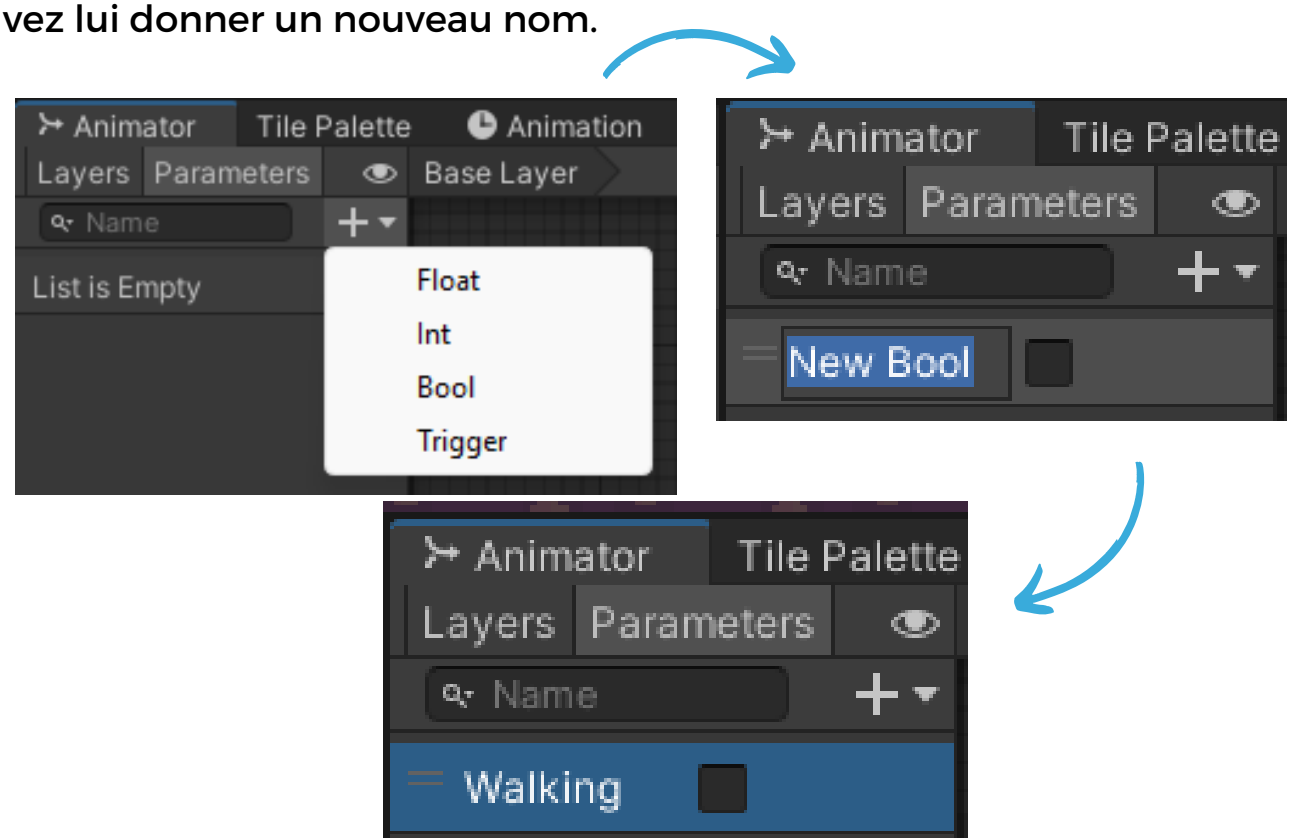
Il faut maintenant créer une condition qui va pouvoir activer cette transition. Nous allons avoir besoin pour cela de paramètres qui pourront être modifiés par le script qui contrôle le personnage.

Paramètres

Afin de créer ces paramètres, rendez-vous à l'onglet "Parameters" de la fenêtre Animator.



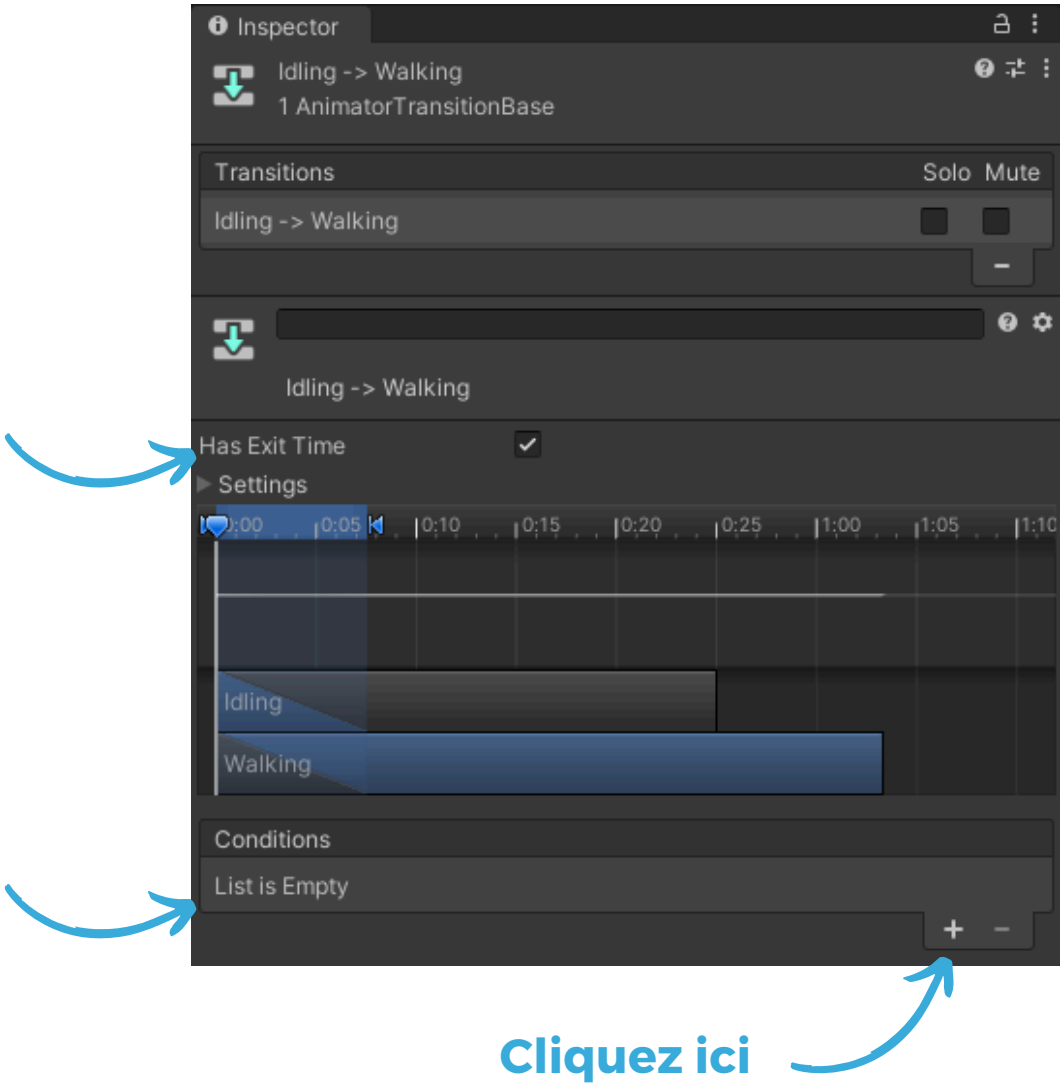
Une fois l'onglet ouvert, cliquez sur le +, un petit menu va s'ouvrir, pour le moment sélectionnez "Bool". Un nouveau paramètre sera ainsi créé, vous pouvez lui donner un nouveau nom.



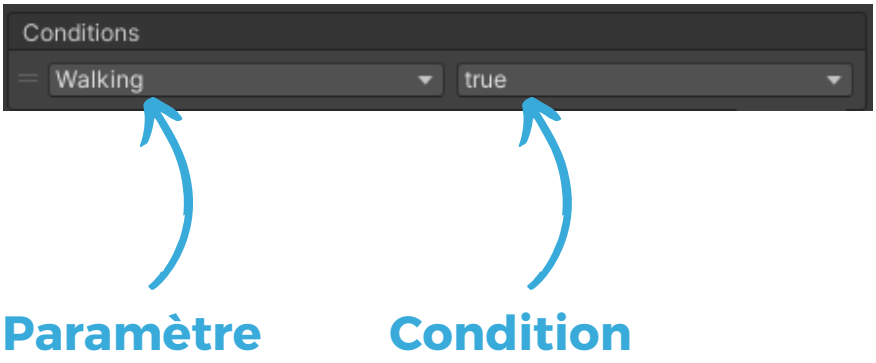
C’est ce paramètre qui va permettre de lancer l’animation mais pour cela il faut d’abord paramétrer la transition adéquatement. Pour ce faire, cliquez sur la flèche de la transition. Dans l’inspecteur vous verrez plusieurs choses utiles.

Tout d’abord la case à cocher “Has Exit Time” (a un temps de sortie) elle correspond au fait que la transition attende ou non que la l’animation précédente soit terminée avant de lancer la suivante. Si elle n’est pas cochée l’animation suivant est lancée dès que la condition est remplie.
Ici comme nous voulons passer d’une animation à l’autre immédiatement elle doit être décochée.

Ensuite les Conditions, elles vont nous permettre de définir quand la transition va s’activer et passer à l’animation suivante.
Pour rajouter une condition, cliquez sur le +.



Nous pouvons voir qu'un nouvelle condition a été créée. Elle comporte deux menus déroulants, un pour choisir le paramètre et un pour définir le moment où la condition est remplie.



Si une des conditions créées est remplie alors la transition s'active.
La seule condition ici étant que le paramètre "Walking" soit true, l'animation sera lancée quand ce dernier sera vrai.

Il va donc falloir que ce paramètre soit vrai quand le personnage marche et faux quand il est immobile. Ou encore qu'il soit vrai que le·la joueur·euse appuie sur les flèches gauche et droite.

De manière assez logique il faut donc aller dans le script du personnage joueur.

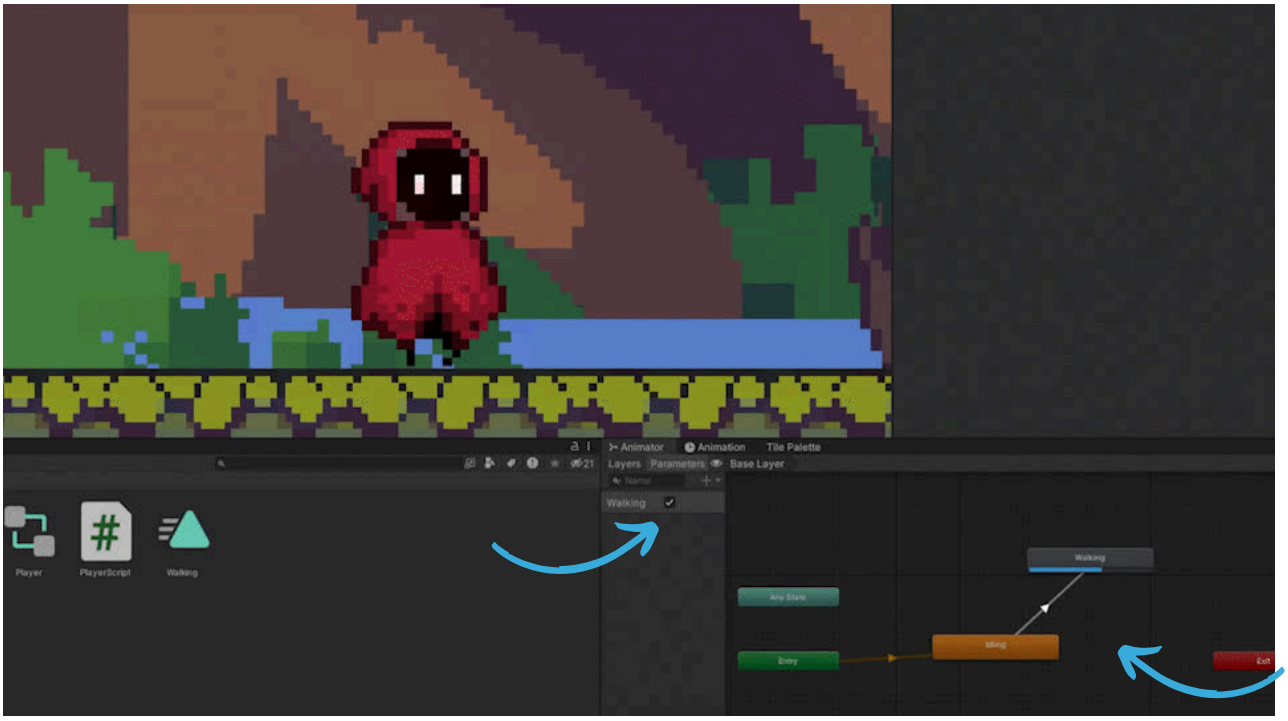
Nous allons tout d'abord devoir créer une variable qui va contenir l'Animator Controller. Cette dernière soit être de type Animator.

```
public class PlayerScript : MonoBehaviour
{
    public float runSpeed;
    public float acceleration;
    public float jumpPower;

    public Rigidbody2D myRigidBody;
    public Animator myAnimator;
    public SpriteRenderer mySpriteRenderer;
```

Vous pouvez voir que j'ai aussi créé une variable de type SpriteRenderer, nous verrons à quoi elle va servir plus tard.

Nous allons donc devoir passer le paramètre booléen “Walking” à true quand le personnage se déplace. Pour ce fairre nous allons utiliser la méthode SetBool() sur notre variable Animator. Cette méthode prend en paramètres le nom du paramètre de l’Animator que nous voulons changer et la valeur (true ou false) que nous voulons lui attribuer.

```
void Update()
{
    if (Input.GetKey(KeyCode.RightArrow))
    {
        transform.position += Vector3.right * Time.deltaTime * runSpeed;
        myAnimator.SetBool("Walking", true);
    }
    if (Input.GetKey(KeyCode.LeftArrow))
    {
        transform.position += Vector3.left * Time.deltaTime * runSpeed;
        myAnimator.SetBool("Walking", true);
    }
    if (Input.GetKeyDown(KeyCode.UpArrow) && canJump)
    {
        myRigidBody.velocity += Vector2.up * jumpPower;
        canJump = false;
    }
}
```



Si nous regardons dans l’éditeur l’onglet Animator en ayant le GameObject du personnage qui est sélectionné nous pouvons voir ce qu’il se passe.
Au démarrage la case du paramètre “Walking” est décochée et l’animation qui est jouée est “Idling”. Ceci se voit avec la barre bleue en dessous de l’animation dans l’onglet Animator.
Quand le·la joueur·euse appuie sur flèche de gauche ou de droite la case du paramètre “Walking” est cochée et l’animation Walking se lance.

Un problème persiste, si le·la joueur·euse arrête d'appuyer sur les flèches l'animation continue d'être jouée. Il manque la transition qui parte de l'animation Walking et aille vers Idling.

Il nous faut donc ajouter quelques lignes dans le script du personnage. Dans le cas où ni la flèche gauche, ni la flèche droite ne sont pressées le paramètre Walking doit être à false.

```
if (!Input.GetKey(KeyCode.RightArrow) && !Input.GetKey(KeyCode.LeftArrow))
{
    myAnimator.SetBool("Walking", false);
}
```

Il va aussi falloir créer une transition de Walking à Idling. Je vous laisse réfléchir à comment paramétrer correctement cette transition.

Les animations devraient maintenant correctement se déclencher.

Il reste un dernier problème, si nous nous déplaçons à gauche le personnage reste tournée vers la droite.

La solution ici est très simple, il faut que le Sprite soit retourné quand l'on va à gauche et non retourné quand l'on va à droite.

Nous allons pour cela modifier la propriété FlipX du SpriteRendere dans le script du personnage.

```
void Update()
{
    if (Input.GetKey(KeyCode.RightArrow))
    {
        transform.position += Vector3.right * Time.deltaTime * runSpeed;
        myAnimator.SetBool("Walking", true);
        mySpriteRenderer.flipX = false;
    }
    if (Input.GetKey(KeyCode.LeftArrow))
    {
        transform.position += Vector3.left * Time.deltaTime * runSpeed;
        myAnimator.SetBool("Walking", true);
        mySpriteRenderer.flipX = true;
    }
```

La solution ici est très simple, il faut que le Sprite soit retourné quand l'on va à gauche et non retourné quand l'on va à droite.

Nous allons pour cela modifier la propriété FlipX du SpriteRenderer dans le script du personnage.

Créer un système de score

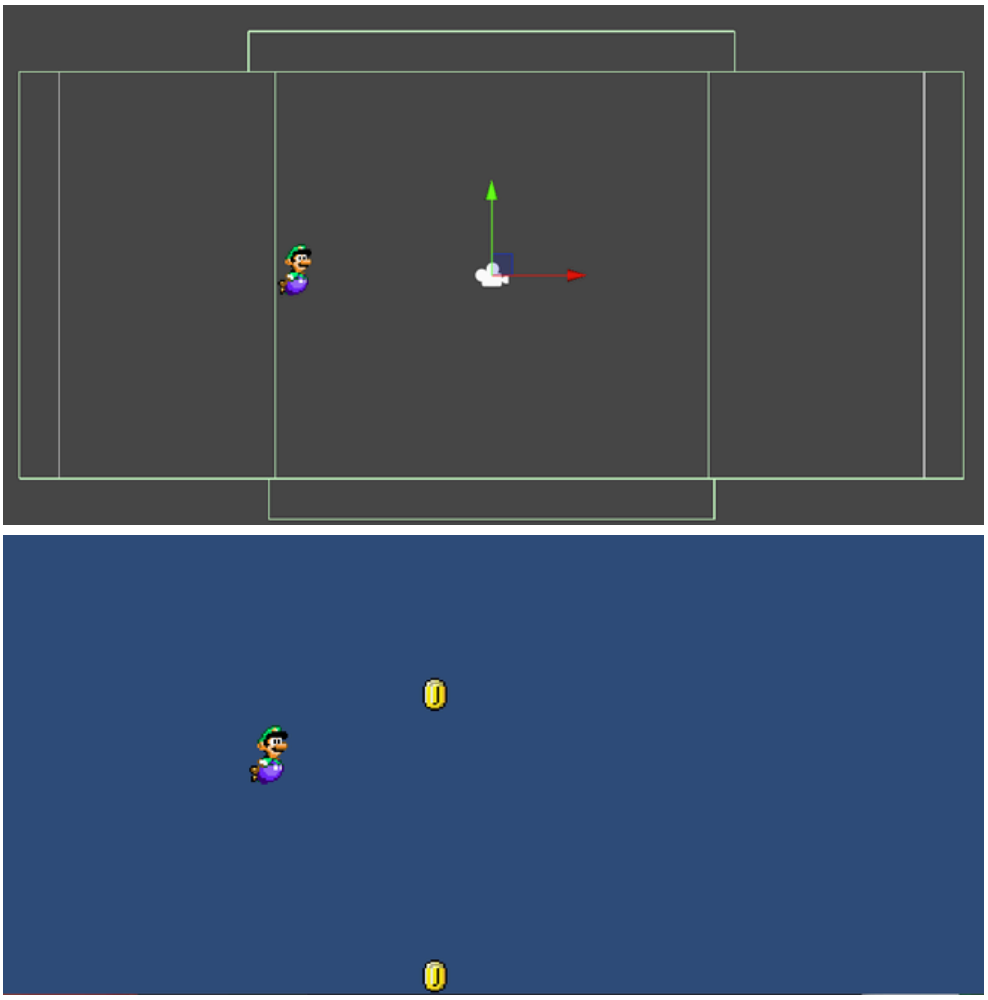
Il est souvent utile de garder des infos générales sur le jeu qui est en train de se dérouler. Si vous voulez par exemple avoir un système de score (un décompte de points qui augmente sous certaines conditions) il va falloir avoir une variable qui va garder en mémoire le score.

Note, je vais utiliser un projet d'un jeu différent que précédemment mais ce dernier n'utilise pas de notions que nous n'avons pas vues. La seule exception étant le système qui fait apparaitre les pièces (pour ça voir "Instantiate", "Destroy", "OnCollision et OnTrigger" et "Layers" dans les fiches).

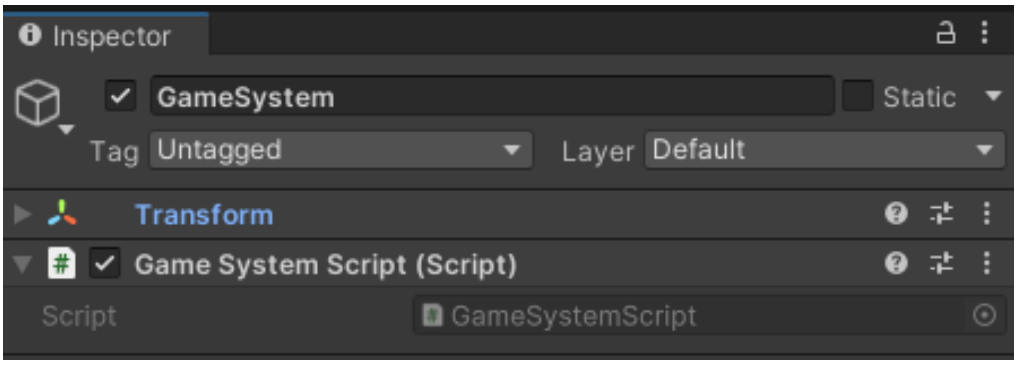
La raison pour laquelle l'explication de la création des pièces se fait dans une autre fiche est simple. Un système de score peut être créé pour compter des objets à récupérer (comme les pièces), des actions à accomplir (découper des fruits dans Fruit Ninja), détruire des ennemis/objets (sauter sur des ennemis ou détruire des blocs dans Super Mario) ou n'importe quoi d'autre.

Le système que je vous présente ici est justement conçu pour être implémenté dans n'importe quel jeu.

Le projet qui va me servir d'exemple comporte un personnage dont on peut contrôler les déplacements de gauche à droite et de bas en haut.
Des BoxColliders forment une zone de laquelle il ne peut pas sortir et des pièces apparaissent à intervalle régulier.



La première étape est de créer un GameObject qui va garder les informations du score. Je l'appelle ici GameSystem mais vous pouvez lui donner le nom que vous voulez. Ce GameObject doit contenir un script qui va contenir les données. Dans la même logique je l'appelle GameSystemScript.



Créons dans ce script une variable qui servira à contenir le score.

```
public class GameSystemScript : MonoBehaviour
{
    int score;
```

Créons maintenant une fonction pour ajouter ou retirer des points au score.

Pour rappel, une fonction est un petit bout de programme auquel on donne un nom et que l'on va pouvoir "appeler" à n'importe quel endroit d'un programme.

Pour créer une fonction il suffit de la déclarer dans un script au même niveau qu'un Update() ou Start().
Il ne faut jamais oublier de définir son type de retour et dans notre cas de changer son scope (voir fiches "Scope").

Pour modifier le score la fonction aura à prendre en paramètre les points à ajouter/retirer et ne renverra rien (elle sera donc de type void).

La déclaration de la fonction devra se faire dans la classe que nous venons de créer.

```
public class GameSystemScript : MonoBehaviour
{
    public int score;

    1 référence
    public void ChangeScore(int change)
    {

    }
```

Dans l'exemple ci-dessus la fonction s'appelle "ChangeScore" et prend un paramètre de type int et nommé "change".

Nous allons donc incrémenter le score de ce changement quand la fonction est appelée. Je vous laisse trouver comment faire et si besoin la solution est juste en dessous.

```
public class GameSystemScript : MonoBehaviour
{
    public int score;

    1 référence
    public void ChangeScore(int change)
    {
        score += change;
    }
}
```

Il est très important que la fonction soit publique (le mot-clé public devant). Ceci a à voir avec le scope que nous avons commencé à voir il y a quelques chapitres.

La fonction ChangeScore va pouvoir être appelée par d'autres scripts, c'est la raison pour laquelle elle doit être publique, sinon elle ne pourrait être appelée que par le script dans lequel elle a été créée (voir fiche "Scope 2").

Maintenant il faut ajouter dans les objets qui influencent le score la fonction ChangeScore. Il faudra l'appeler quand les condition de récupération de points sont remplies.

Dans mon exemple quand le personnage touche une pièce, la pièce doit disparaitre et le score doit monter d'un point.

Mais pour pouvoir appeler une fonction dans un autre script il va falloir référencer ce script. Pour ce faire, nous allons créer dans l'objet qui doit appeler la fonction une variable du type du script. Dans mon exemple, je dois donc créer une variable de type GameSystemScript dans le script de mes pièces.

```
public class CoinSCript : MonoBehaviour
{
    public GameSystemScript gameSystem;
```

Cette variable va récupérer le script du game system quand elle apparait. Pour cela nous allons utiliser la fonction FindAnyObjectByType qui va chercher dans les objets qui sont présents dans le jeux le premier qui a un composant du type donné.

```
void Start()
{
    gameSystem = FindAnyObjectByType<GameSystemScript>();
}
```

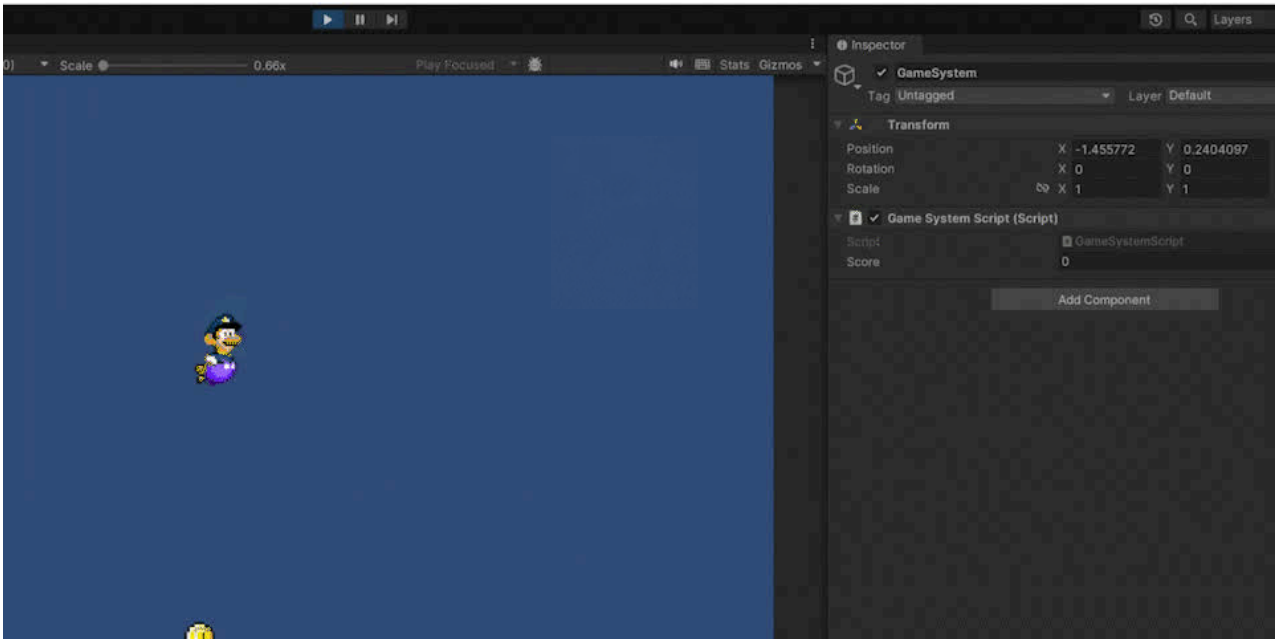
Il faut bien faire attention à ce qu'il n'y ai qu'un seul GameObject avec un GameSystemScript.

Il ne reste plus qu'à utiliser cette référence au script pour appeler la fonction ChangeScore quand une pièce est touchée. Pour ce faire il suffit d'utiliser le nom de la variable qui contient le script suivi d'un point et de la fonction à appeler.

Nous allons donc utiliser le fonction OnTriggerEnter2D.

Dans la vidéo ci-dessous la pièce disparait quand elle est touchée, nous verrons plus tard comment en arriver là.

```
Message Unity | 0 références
private void OnCollisionEnter2D(Collision2D collision)
{
    gameSystem.ChangeScore(1);
```



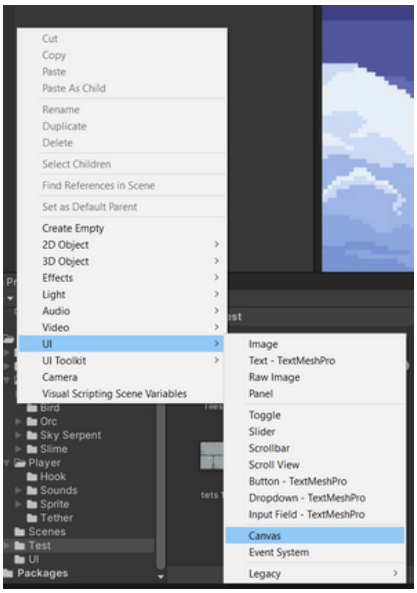
Nous pouvons voir que notre variable score augmente bien quand on touche une pièce. Il nous reste maintenant à afficher ce score à l'écran.

Pour ce faire nous allons créer des éléments d'UI (User Interface ou interface utilisateur).

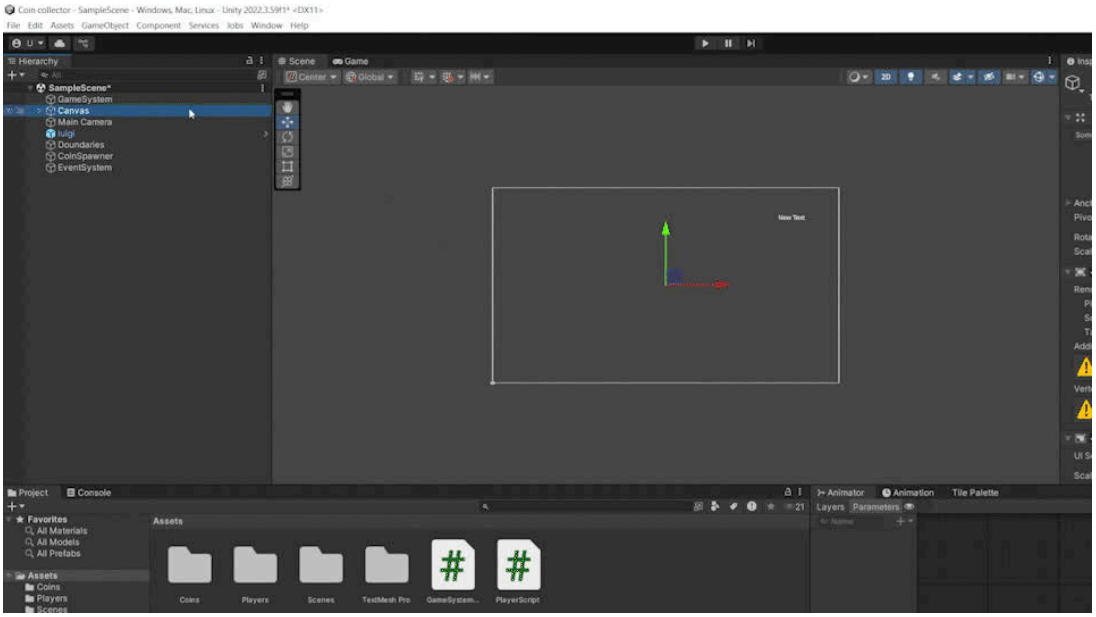
Heureusement pour nous Unity inclut des éléments d'UI pré faits et faciles d'utilisation.

Avant toute chose nos éléments d'UI ont besoin d'être placés dans un GameObject particulier, le Canvas.
Ce dernier est un GameObject qui va contenir tous les éléments d'UI.

Pour le créer, faites clique droit dans votre hiérarchie, sélectionnez UI puis Canvas.

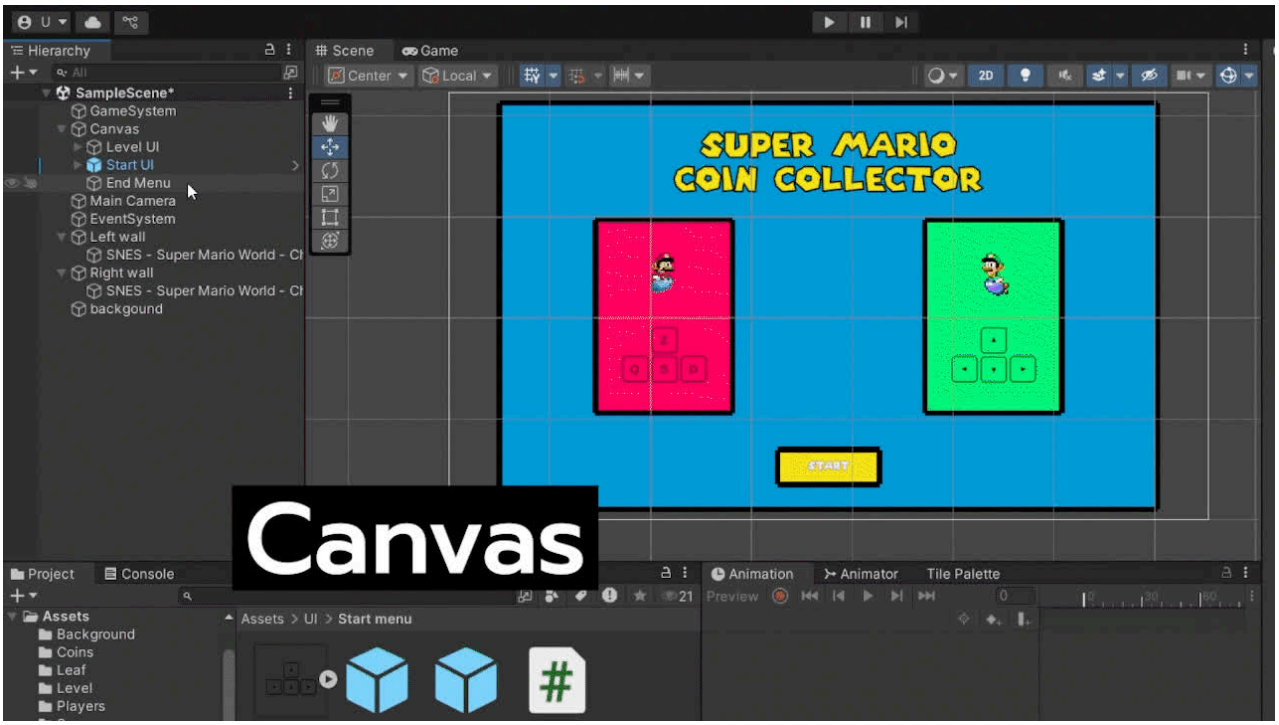


Attention que le Canvas est un très grand objet. Pour voir le Canvas dans son entièreté, double-cliquez dessus.



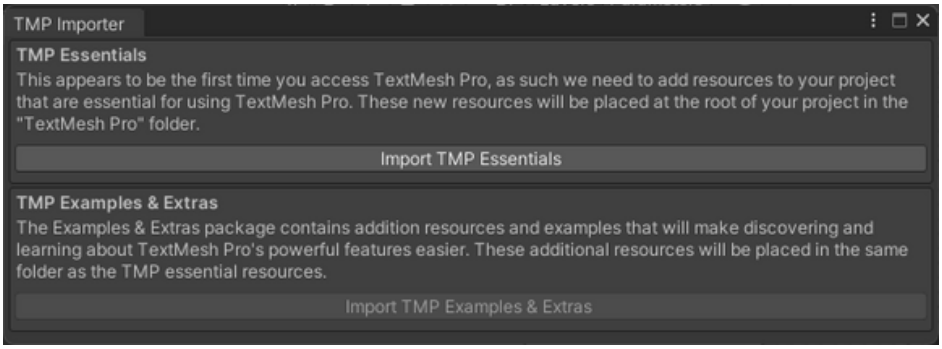
Nous allons voir une chose importante avec le Canvas et les éléments d'UI. Le Canvas est généralement beaucoup plus grand que les éléments de votre jeu.

Le Canvas délimite une grande zone rectangulaire et les éléments que vous allez mettre dans le Canvas seront visibles à l'écran en jeu en fonction de leur position dans le Canvas.

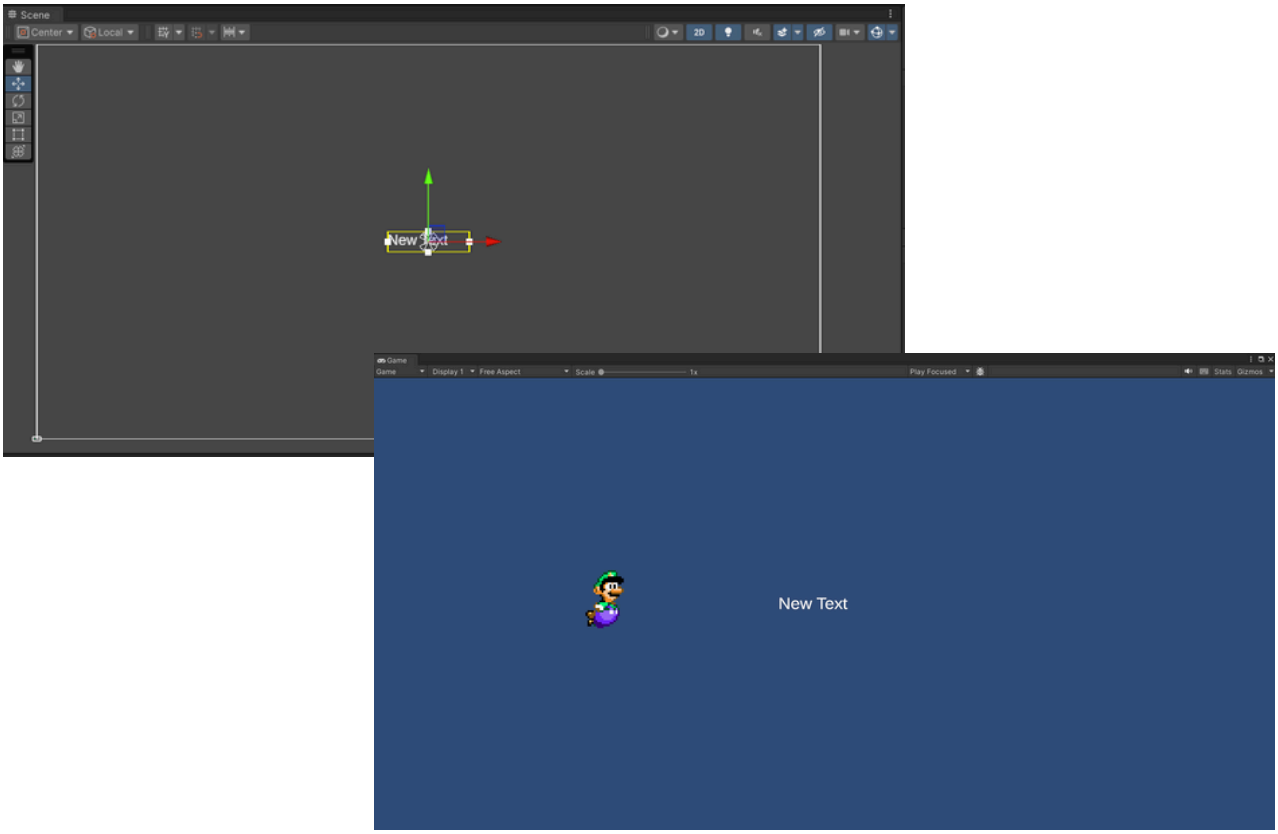


Maintenant, créons un élément qui va permettre d'afficher du texte pour afficher notre score. L'élément qui nous faut est un TextMeshPro, faites donc un clic droit sur le Canvas dans la hiérarchie, sélectionnez "UI" puis "Text - TextMeshPro".

La première fois que vous créez un TextMeshPro vous devrez importer un module, pour ce faire, cliquez simplement sur Import TMP Essentials.



Une fois le TextMeshPro créé vous pourrez le voir dans le Canvas et dans le jeu dans la fenêtre Game.

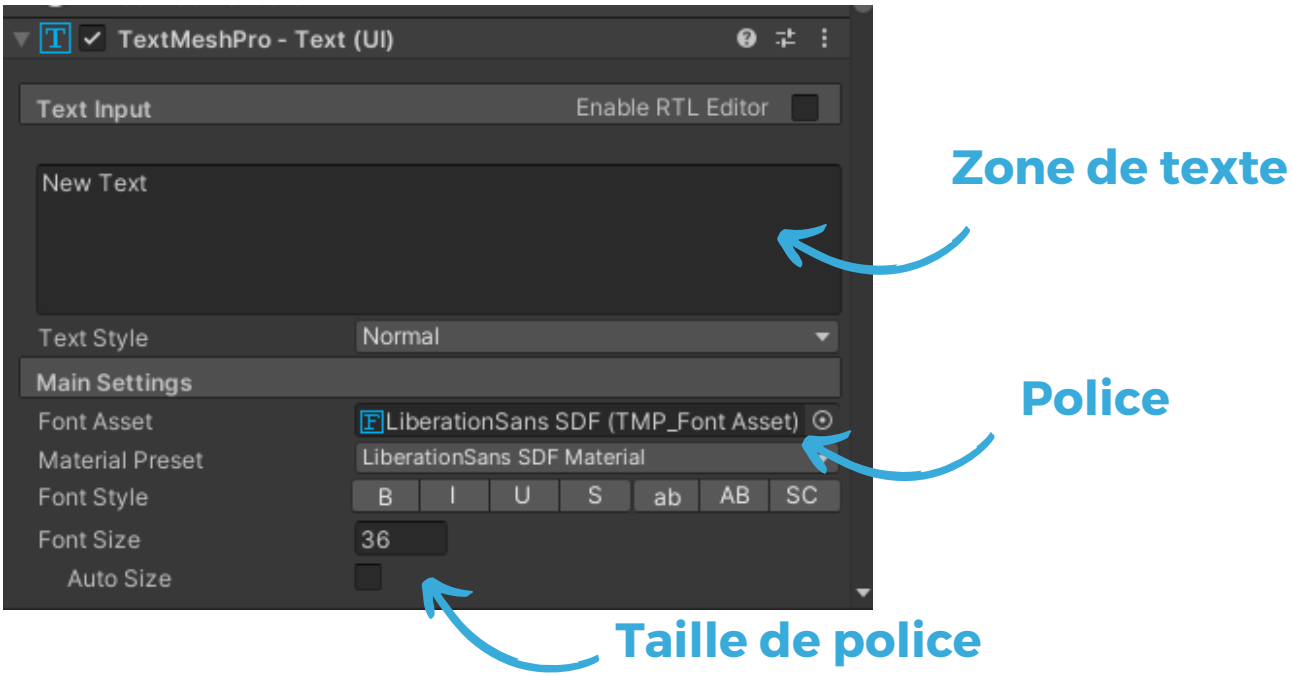


Vous pouvez déplacer le TextMeshPro dans le Canvas pour changer sa position dans l'écran du jeu. Pour ce faire, sélectionnez-le, cliquez sur l'outil "Move" en haut à gauche de la scène et utilisez les flèches verte et rouge.

Pour plus d'informations sur les TextMeshPro, rendez-vous à la fiche du même nom.

Vous pouvez modifier le texte du TextMeshPro et sa police en le sélectionnant et en allant dans l'inspecteur. Vous trouverez un composant "TextMeshPro - Text (UI)".

Il comporte une zone de texte qui sera le texte affiché. Il est aussi possible de choisir une police (Font Asset) et une taille de police (Fonte Size). Nous verrons plus tard comment importer une police.



Il va maintenant falloir être capable de changer le texte affiché via un de nos script.

Dans mon exemple le plus logique serait de changer le texte à chaque changement de score, donc dans la fonction ChangeScore.

Pour cela il nous faudra référencer le TextMeshPro dans notre script. Attention, la variable devra être de type TextMeshProUGUI (attention à ne pas confondre avec TextMeshPro et TextMesh).

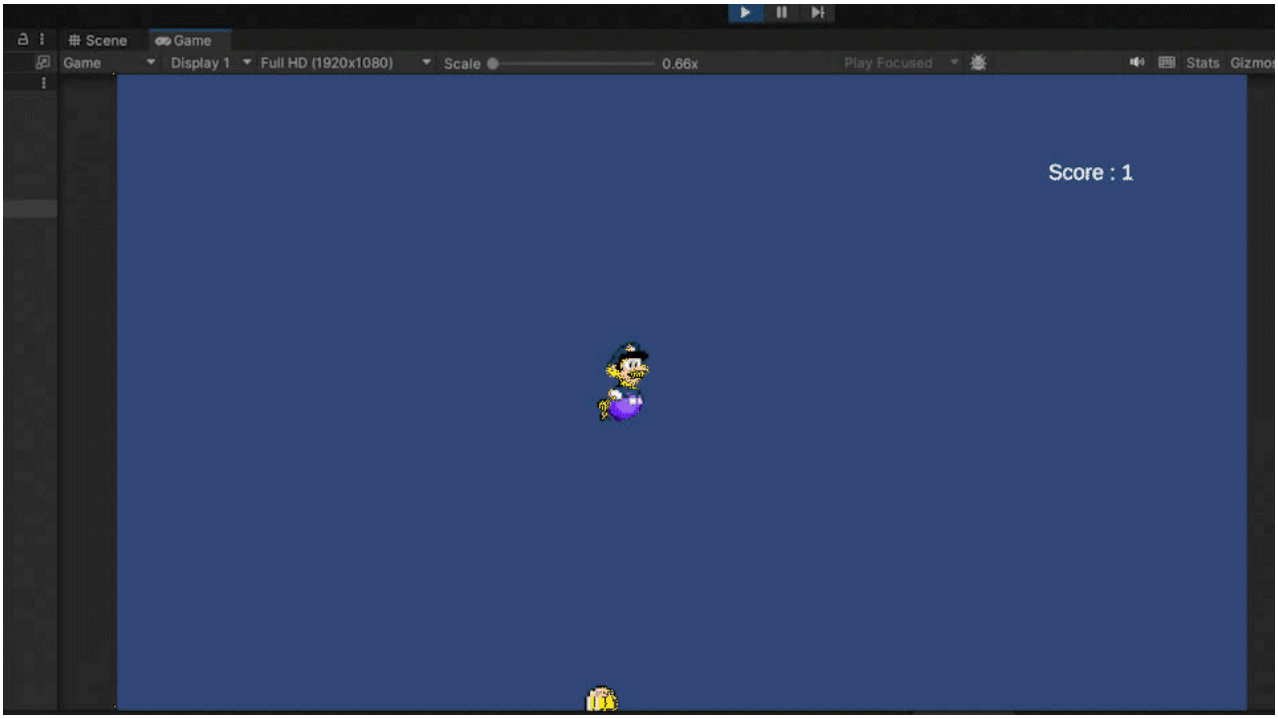
```
public class GameSystemScript : MonoBehaviour
{
    public int score;

    TextMeshProUGUI scoreText;
```

N'oubliez pas d'assigner le TextMeshPro à la variable que vous venez de créer.

Maintenant il suffit d'ajouter dans la fonction ChangeScore une ligne qui va changer le texte du TextMeshPro.

Pour ça, rien de plus simple il suffit d'utiliser la variable qui contient le TexteMeshPro par son nom et d'accéder à sa propriété text. Il faudra donc assigner le nouveau texte à cette propriété.

```
public void ChangeScore(int change)
{
    score += change;
    scoreText.text = "Score : " + score;
}
```



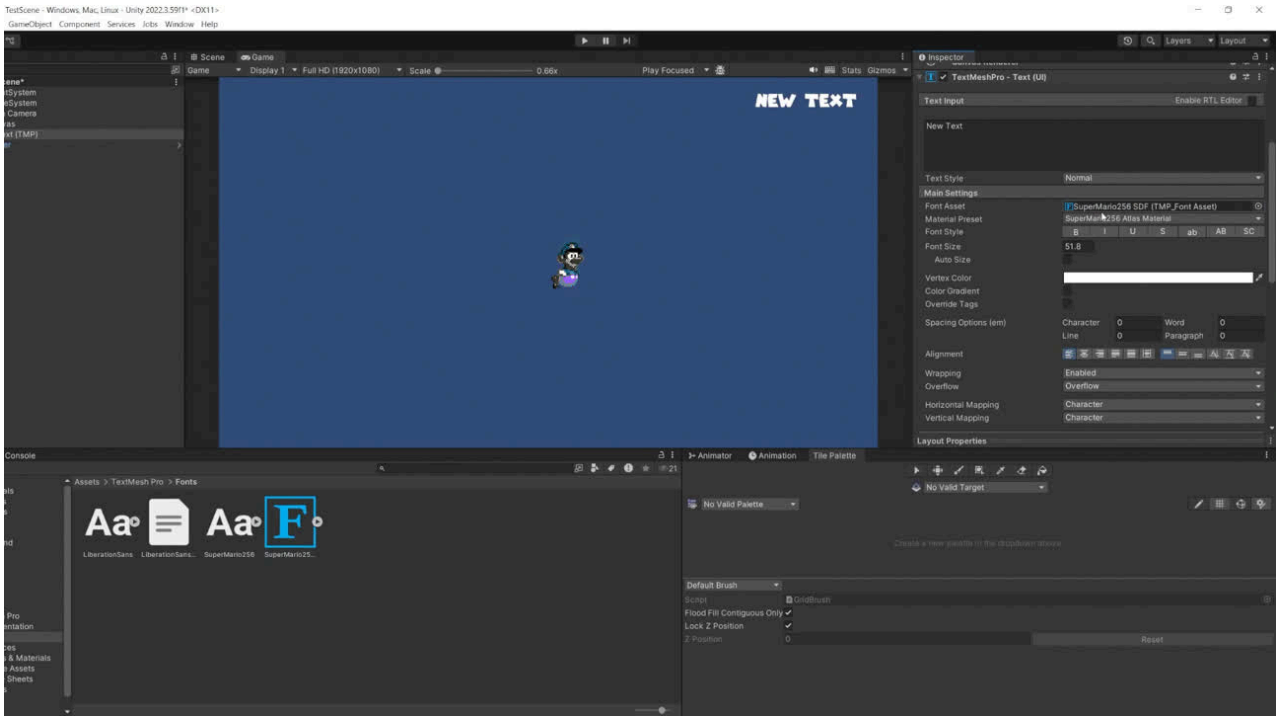
Améliorer l'UI

Vous savez maintenant créer un élément d'UI basique (un score affiché avec du texte). Mais ce dernier va probablement jurer avec votre jeu, son design ne correspondra pas au style de votre jeu.

Pour remédier à ce problème il va falloir aller plus dans le détail et personnaliser ce score.

La première étape pourrait être de changer la police avec laquelle le score est écrit. Dans un premier temps cherchez et importez la police de votre choix. Pour ce faire référez-vous à la fiche "Importer des polices".

Une fois votre police importée il vous suffit de glisser-déposer la police dans la case correspondante du TextMeshPro.

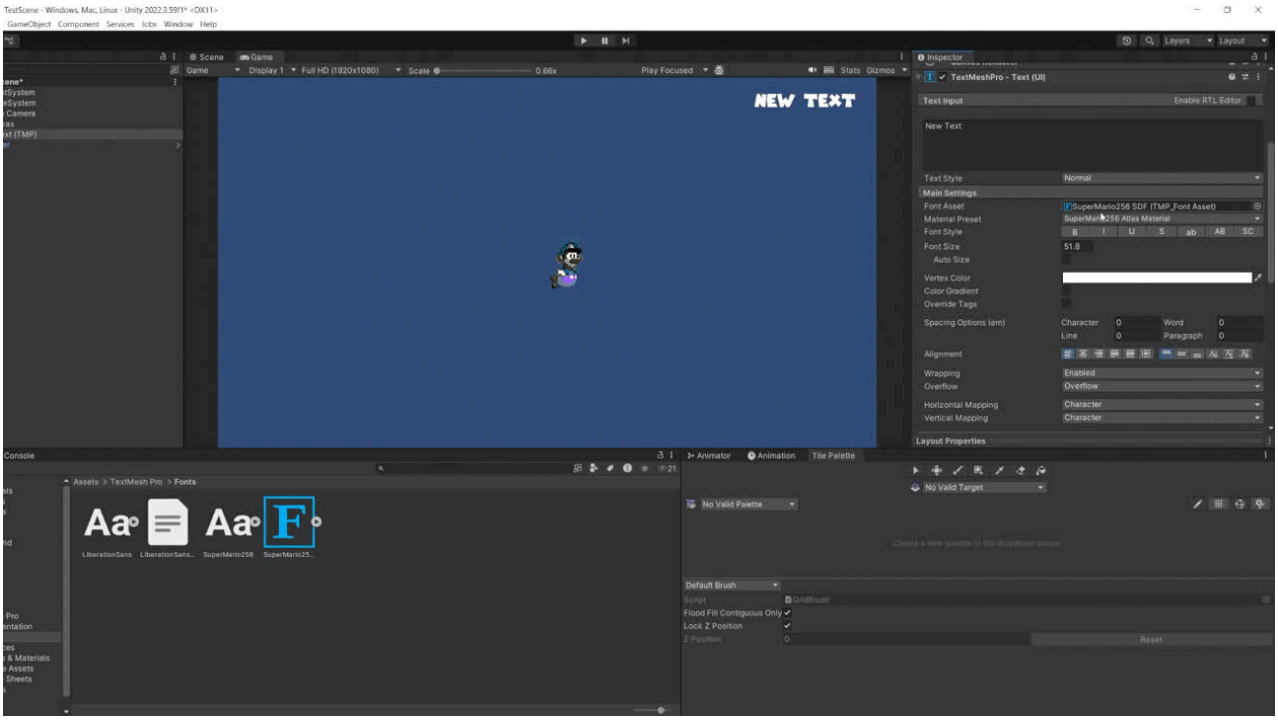


Vous savez maintenant créer un élément d'UI basique (un score affiché avec du texte). Mais ce dernier va probablement jurer avec votre jeu, son design ne correspondra pas au style de votre jeu.

Pour remédier à ce problème il va falloir aller plus dans le détail et personnaliser ce score.

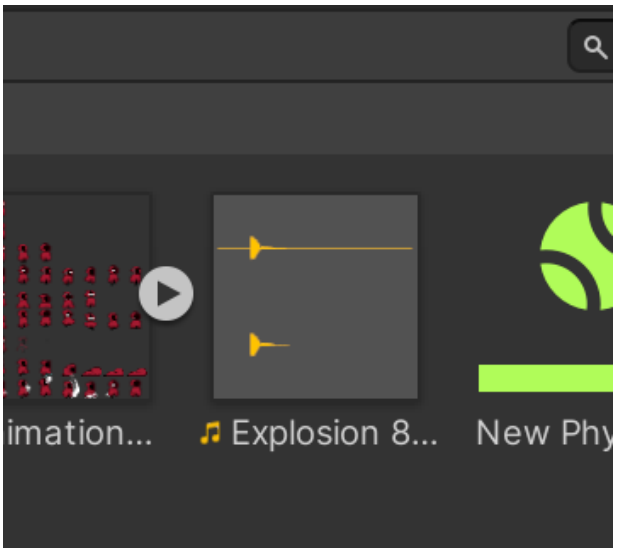
La première étape pourrait être de changer la police avec laquelle le score est écrit. Dans un premier temps cherchez et importez la police de votre choix. Pour ce faire référez-vous à la fiche "Importer des polices".

Une fois votre police importée il vous suffit de glisser-déposer la police dans la case correspondante du TextMeshPro.

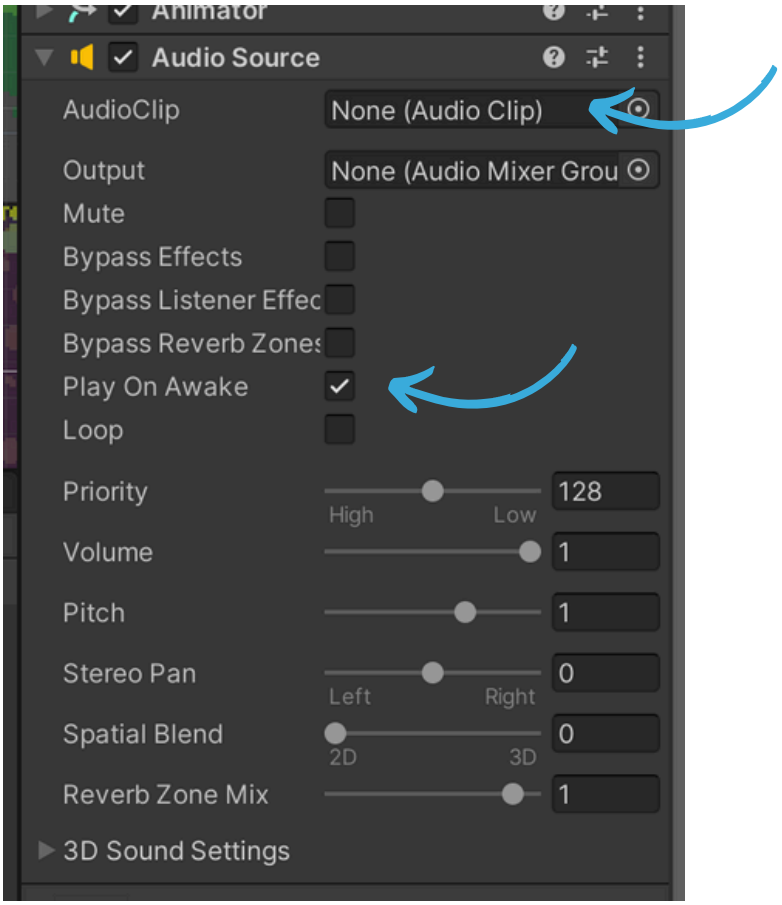


Ajouter des sons

Commencez par importer un fichier mp3, vous obtiendrez un Audio Clip.



Pour jouer un son il faut qu'un gameobject dispose du composant "AudioSource". Ce dernier va pouvoir jouer un audioclip que l'on aura sélectionné. Une fois l'Audiosource ajouté, glissez l'Audioclip dans la propriété correspondante et décochez "Play on Awake".



Rendez-vous maintenant dans le script du GameObject que vous voulez bruiter et créez une variable public de type AudioSource.

```
public AudioSource myAudioSource;
```

Dans votre script, quand vous voudrez jouer le son, utilisez la commande .Play sur votre AudioSource (le nom de la variable qui contient l'AudioSource suivi de .Play()).

```
myAudioSource.Play();
```

Et finalement n'oubliez pas d'attribuer le composant AudioSource à la variable dans l'éditeur.