Recréer Pong en 3D avec Unity et C#

Introduction à Unity

Unity est un moteur de jeu très populaire qui permet de développer des jeux pour toutes les plateformes (PC, MAC, Linux, Consoles, Android, iOS, Web…). Vous développez votre projet une fois et vous pouvez l'exporter pour n'importe quelle plateforme. C'est un atout majeur et un gain de temps considérable. La création de jeux se fait sous Unity (pour la partie conception des niveaux, paramétrages, assemblage des éléments…) et sous un éditeur de code comme Visual Studio pour la partie développement. Le langage de programmation utilisé dans Unity est le C#. Il s'agit d'un langage plus simple à apprendre que le C++ car la syntaxe est moins lourde.

Unity est un éditeur WYSIWYG (What you see is what you get). En d'autres mots, tout ce que vous créez dans Unity (par exemple un niveau) sera visible tel quel par le joueur final. Cela est bien pratique car vous pouvez tester vos niveaux à la volée et savoir quel sera le rendu final sur l'écran de l'utilisateur. Le tout se passe dans une interface intuitive et assez simple à prendre en main.

Malgré tout, Unity existe depuis 2005 environ (le logiciel existait déjà avant mais sous un autre nom et non accessible au grand public) et dispose aujourd'hui de très nombreuses fonctionnalités. La principale difficulté dans l'apprentissage de Unity est donc de s'y retrouver dans la quantité d'informations et d'outils.

Ce cours a pour but de vous accompagner dans la prise en main de Unity de façon simple en utilisant des fonctionnalités basiques afin de démarrer en douceur. Les quelques fonctionnalités que nous apprendrons vous seront cependant utiles pour l'ensemble de vos futurs projets car je me concentrerai sur ce qui est le plus utilisé pour créer des jeux. Pour découvrir comment créer des jeux avec Unity, je vous propose de développer une sorte de clone de Pong en 3D avec une caméra fixe. Cela nous permettra d'apprendre en nous basant sur un projet concret.

Enfin, pour terminer cette petite présentation, je précise que si votre but est de créer un jeu AAA comme GTA, Fortnite, Warcraft, Hogwarts legacy ou autre, vous n'êtes pas au bon endroit. En effet ces jeux sont développés durant des années, par des centaines (ou milliers) de développeurs et nécessitent des millions de dollars d'investissement tout au long du développement. Il n'est pas possible de développer de tels jeux tout seul ou en petite équipe. A notre niveau, ce que nous pouvons faire se sont de petits jeux indépendants, principalement à destination des mobiles car c'est ce qui reste le plus simple. En revanche, après avoir créé plusieurs petits projets, après avoir acquis une forte expérience et après avoir fait vos preuves, vous pourrez trouver un emploi dans le secteur et potentiellement travailler sur un gros projet dans le futur.

Cela étant dit, commençons par télécharger Unity.

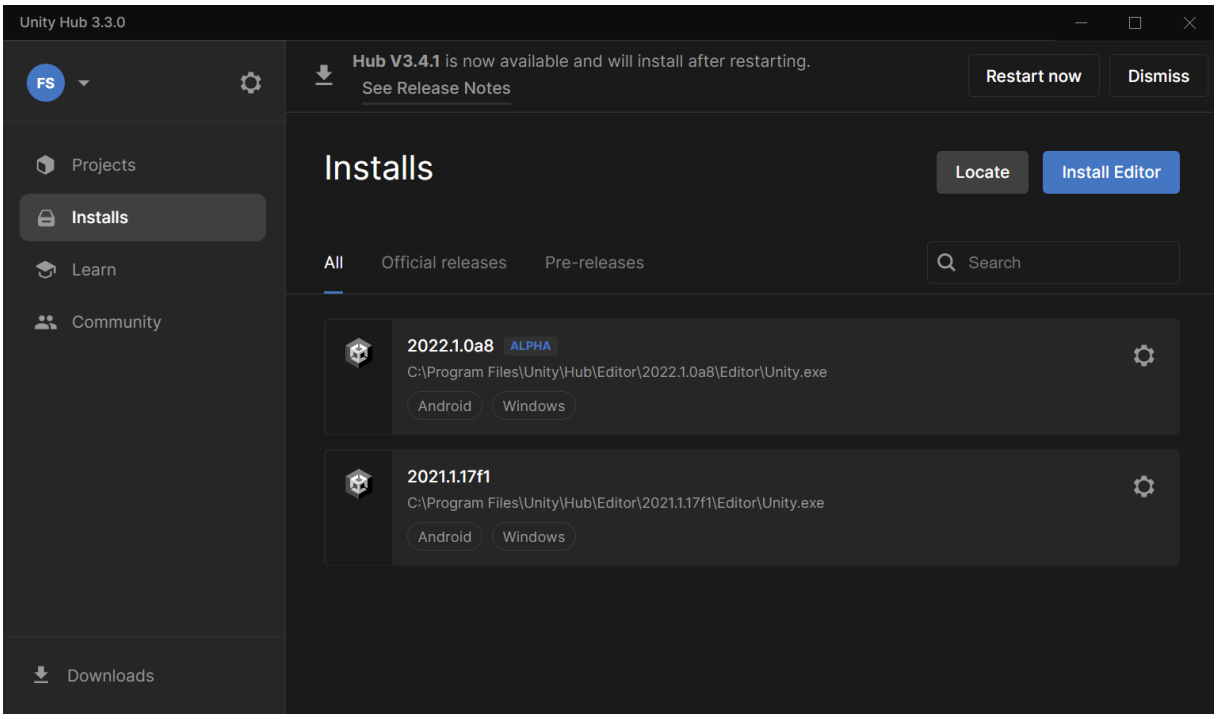
Premier lancement de Unity et installation des outils

Avant toute chose, nous devons télécharger le logiciel afin de pouvoir l'installer. Pour cela, je vous invite à vous rendre sur le site officiel https://unity.com/fr et à vous rendre dans la section téléchargement (https://unity.com/fr/download) afin de télécharger la version de Unity HUB qui correspond à votre système d'exploitation.

Unity HUB est un petit utilitaire qui vous permet de télécharger Unity et de gérer différentes versions du logiciel. Vous pouvez en effet avoir plusieurs versions du logiciel installées en simultané sur votre ordinateur. Le HUB vous permettra de lancer vos différents projets avec la bonne version de Unity.

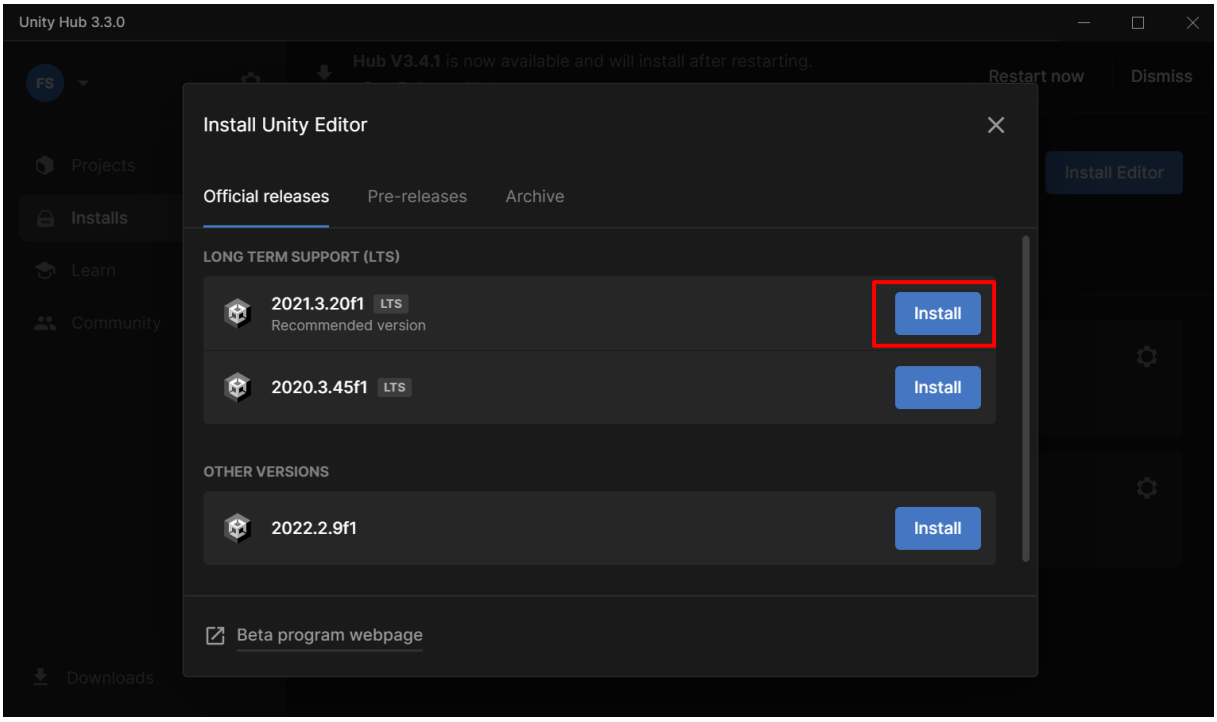
Unity est un logiciel gratuit (tant que vous réalisez un chiffre d'affaires de moins de 100 000 $ par an). Pensez bien à choisir la version « Personnal » lorsque vous aurez à choisir votre situation. La version personnal est gratuite mais nécessite de créer un compte sur le site de Unity pour pouvoir l'activer. De plus, votre compte vous permettra aussi de télécharger des ressources sur l'assetstore (magasin de Unity qui regroupe des milliers de modèles 3D, textures, musiques etc.).

Une fois Unity HUB installé, vous retrouverez sur la gauche de la fenêtre le menu « Installs » qui vous permettra d'installer Unity :



Cliquez sur « Install Editor » afin de choisir la version de Unity que vous souhaitez installer. Je vous recommande d'installer la version LTS la plus récente. Les versions LTS sont celles qui disposent de mises à jour sur le long terme.

Depuis Unity HUB vous avez accès à toutes les versions de Unity, les anciennes, les nouvelles mais aussi les versions en alpha ou bêta. Évitez d'utiliser des pré-versions pour de vrais projets. Ces versions en cours de développement permettent juste de tester et d'apprendre en avance mais ne sont pas faites pour être utilisées pour de la production. Cliquez sur le bouton « Install » qui correspond à la version que vous souhaitez installer :



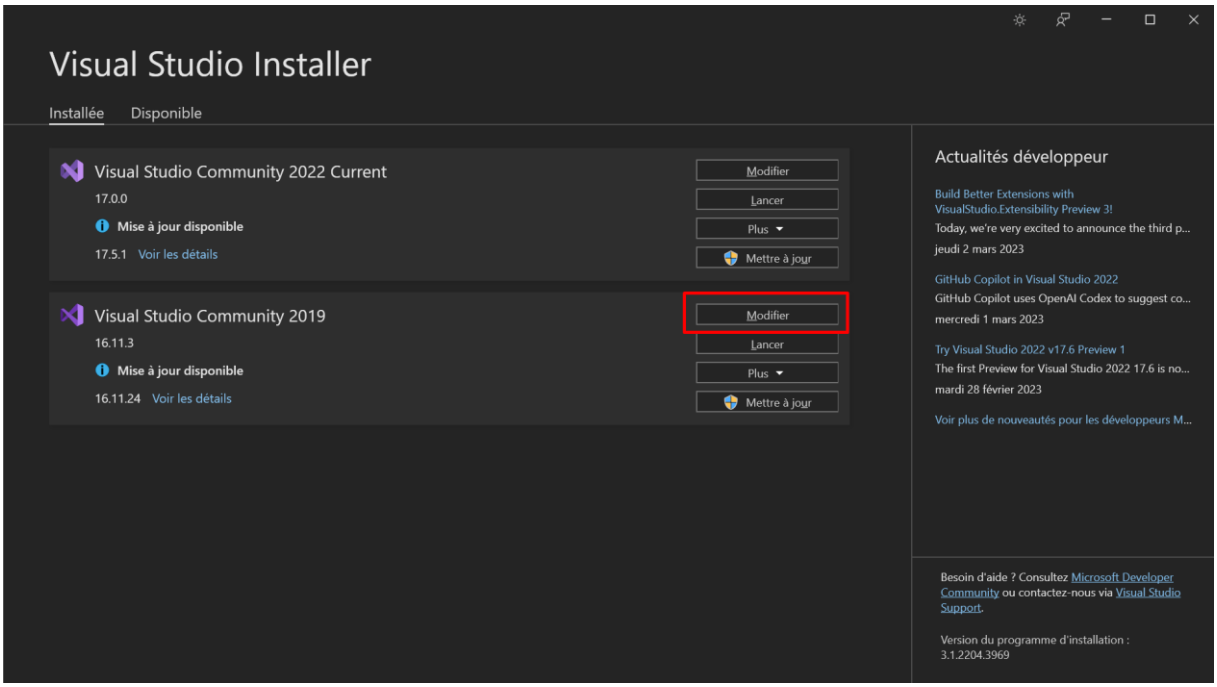
Sachez que ce qui est présenté dans ce cours devrait marcher peu importe la version que vous utilisez. Donc si au moment où vous lisez ce cours la version LTS est différente, cela ne devrait pas poser de problème.

Lorsque vous cliquez sur « Install » une popup s'ouvre et vous invite à sélectionner les fonctionnalités que vous souhaitez installer. Vous pouvez par exemple installer les outils de développement pour mobiles, consoles, TV connectées, navigateurs etc. Dans un premier temps, je vous recommande d'installer uniquement la version standard (qui fait déjà plus de 5 Gb) et d'installer Visual Studio Community afin de pouvoir développer avec le langage de programmation C#. Visual studio est indispensable pour bien suivre ce cours. Si jamais vous rencontrez des difficultés avec l'installation de Visual Studio, vous pouvez le télécharger et l'installer manuellement en téléchargeant l'installeur sur le site officiel (https://visualstudio.microsoft.com/fr/downloads/).

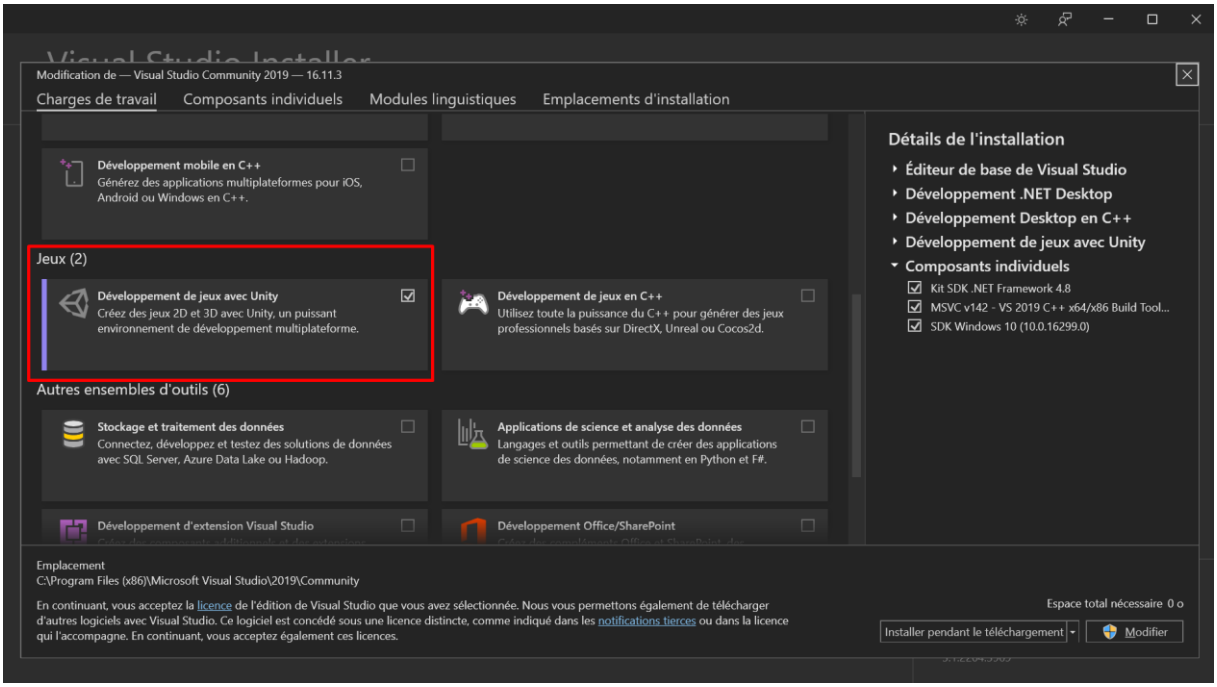
Une fois les options cochées, cliquez sur « Install » pour poursuivre et allez jusqu'au bout du processus d'installation.

Avant de poursuivre, pour être sûr que Visual Studio est correctement installé, je vous invite à ouvrir Visual Studio Installer. Si vous n'avez pas cet utilitaire, vous pouvez le retrouver sur le site de Microsoft : https://visualstudio.microsoft.com/fr/downloads/.

Comme pour Unity HUB, Visual Studio Installer est un utilitaire qui vous permet de gérer les différentes versions du logiciel. Vérifiez que le logiciel est installé et cliquez sur le bouton « Modifier » afin d'afficher les fonctionnalités activées :

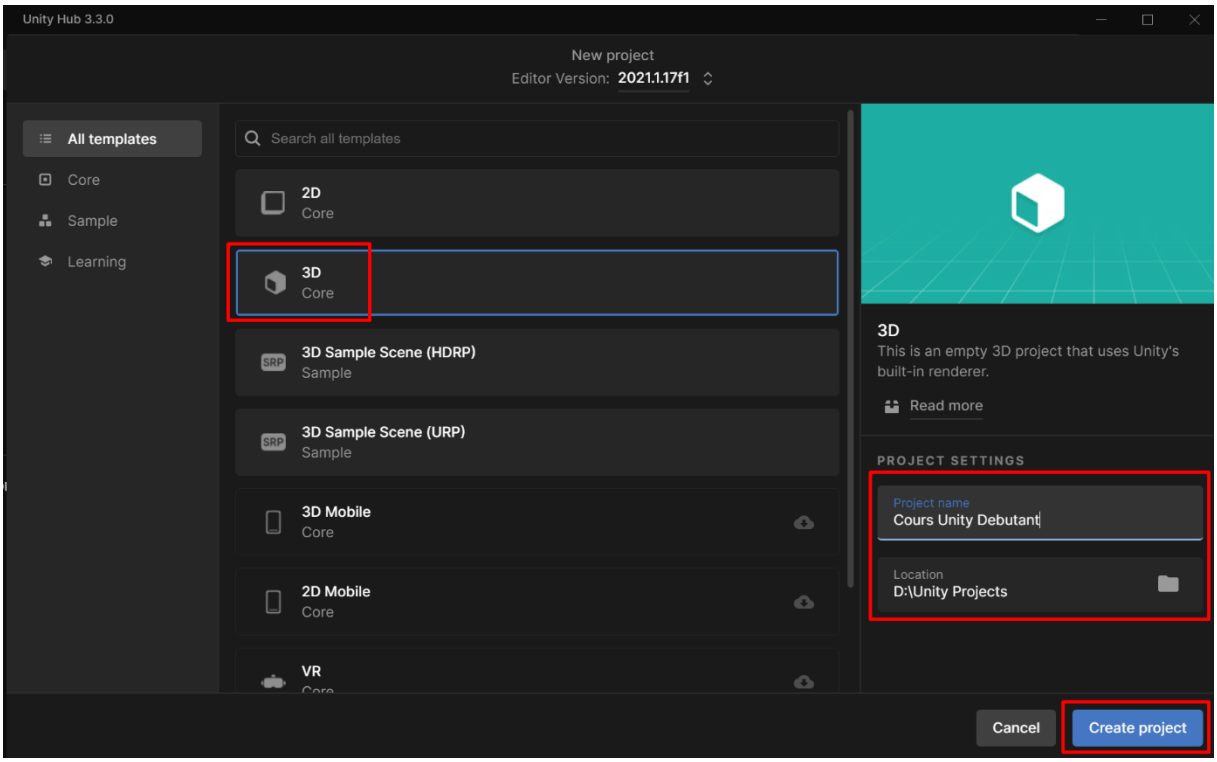


Ici, ce que je souhaite que vous vérifiiez, c'est que le développement de jeux avec Unity est bien installé :

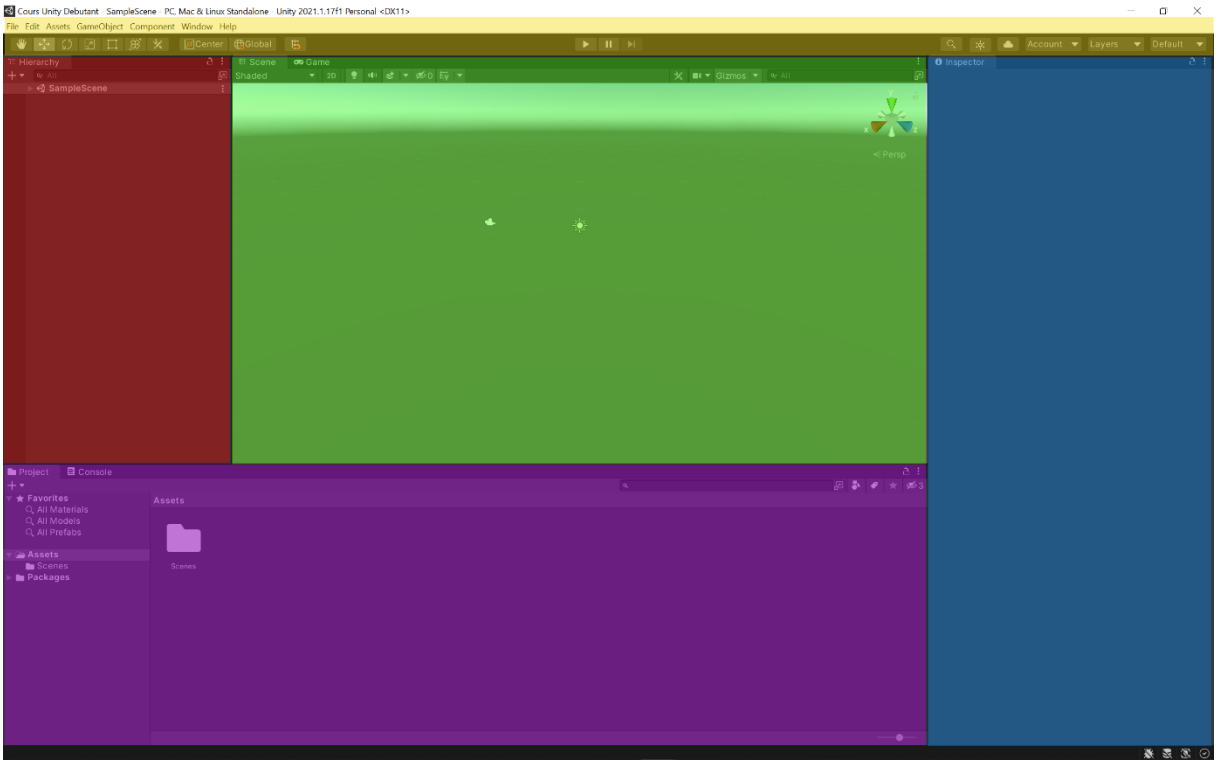


Si ce n’est pas le cas, faites le nécessaire. Cela permettra de programmer plus efficacement grâce à un assistant intelligent.

Retournons sur Unity HUB et cliquez sur « New project » afin de créer un nouveau projet. Choisissez le type de projet « 3D », donnez un nom à votre projet, sélectionnez un emplacement puis cliquez sur « Create project » pour valider :



Le projet va alors se créer et Unity Editor devrait s’ouvrir. Je vous propose de regarder rapidement l’interface du logiciel pour comprendre comment celle-ci est découpée :



Avant tout, sachez que toutes les zones de l'interface peuvent être redimensionnées, déplacées, masquées, ajustées. Vous pouvez donc adapter l'interface à vos besoins. Je vais essayer cependant de garder l'interface par défaut afin que ce soit plus facile pour vous pour débuter. J'ai découpé les principales zones par couleur :

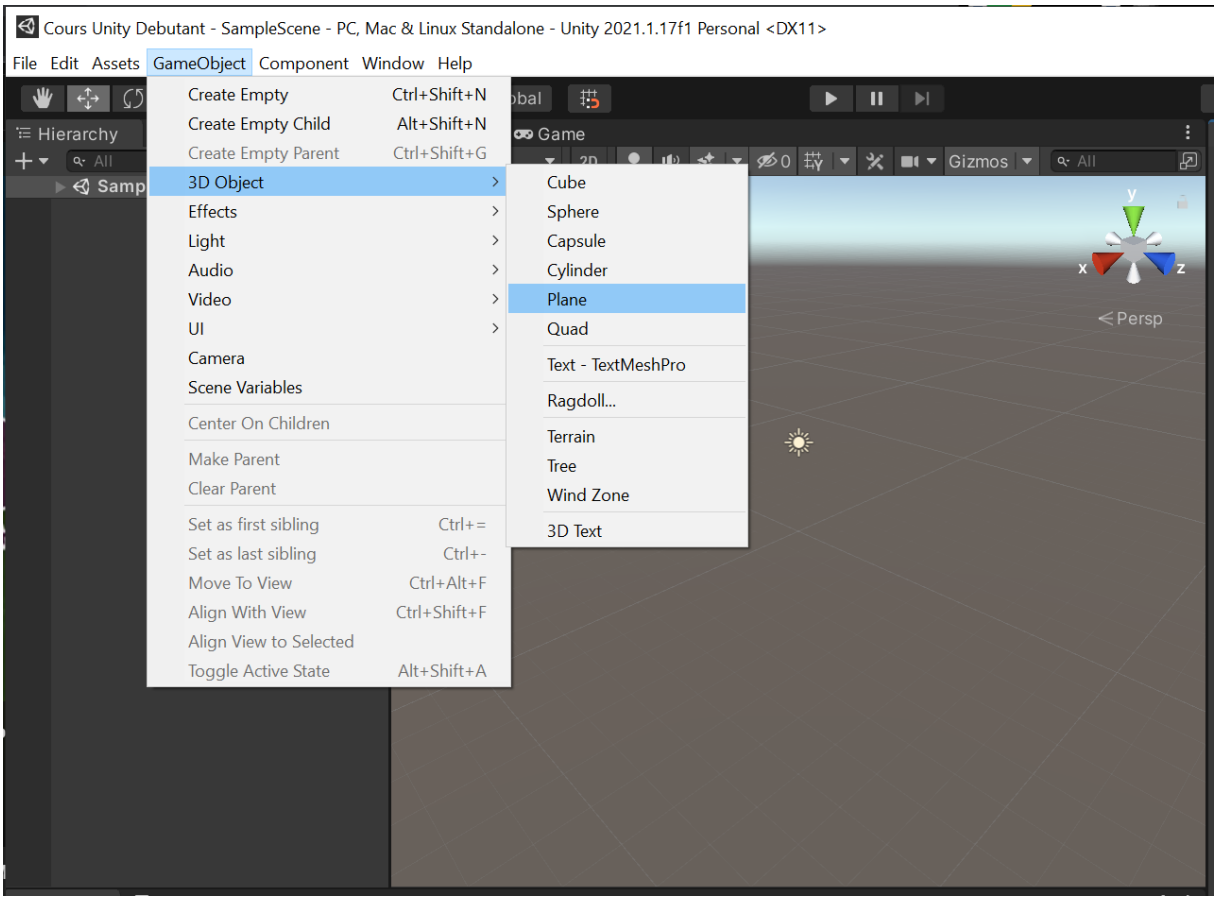
- En jaune, vous avez le menu et la barre d'outils. Vous y retrouverez quelques raccourcis dont certains qui permettent de manipuler les modèles 3D et d'autres qui permettent de tester le jeu à la volée directement dans Unity.
- En rouge vous avez la hierarchy (J'utiliserai toujours le mot anglais car Unity n'existe pas en français). Il s'agit de la liste de tous les éléments composant votre scène.
- En vert la scene (encore une fois en anglais donc sans accent). C'est là que vous allez concevoir vos niveaux, placer les objets, manipuler les assets (ressources). Vous avez également un système d'onglets permettant de basculer de la scene au game. Game est la fenêtre correspondant au rendu final de votre jeu tel que le verra le joueur. C'est dans cette zone que vous pourrez tester le projet.
- En bleu l'inspector. Lorsque vous sélectionnerez un objet sur la scène, vous verrez l'ensemble de ses propriétés et caractéristiques directement dans l'inspector.
- En violet le project. C'est ici que vous retrouverez toutes les ressources (assets) que vous aurez importées dans Unity. Ces ressources seront utilisables pour

concevoir votre jeu. Dans un onglet séparé vous aurez la console qui affichera les différentes erreurs de votre code afin de vous aider à les corriger.

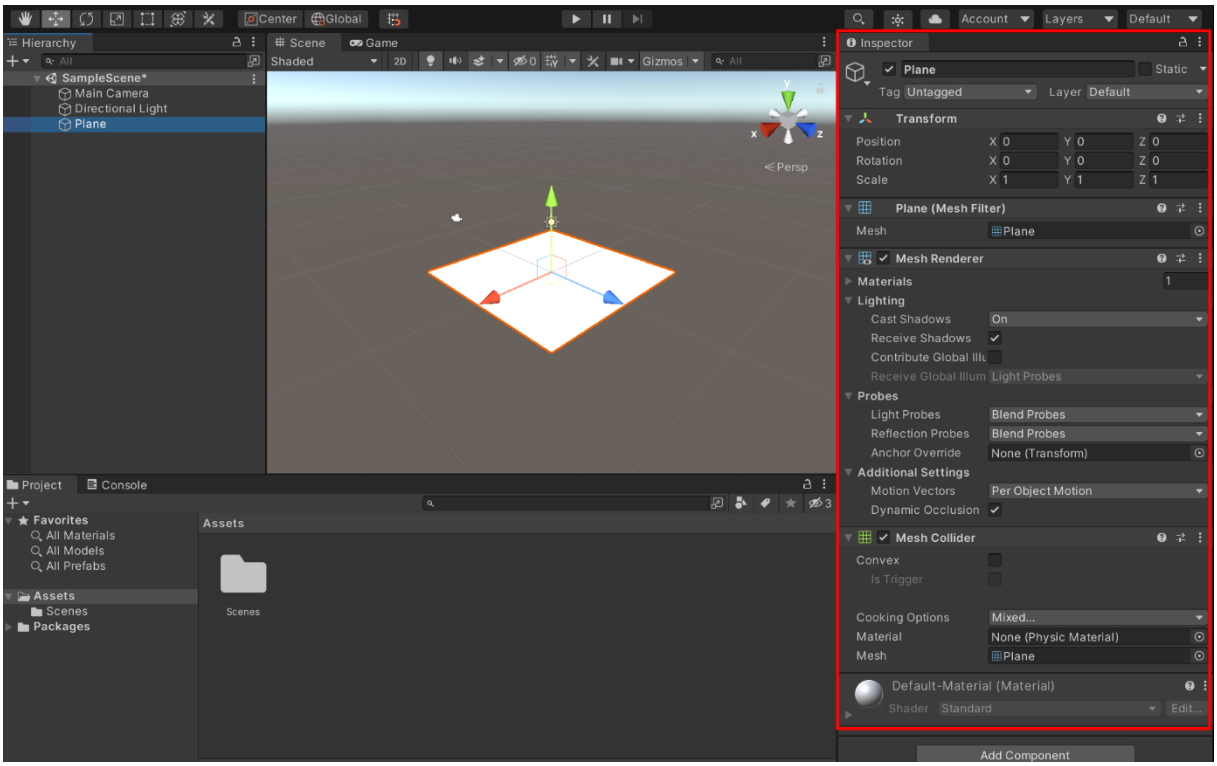
Voila rapidement ce qui compose l'interface du logiciel. Nous aurons l'occasion de découvrir plus en détail les différentes zones au moment où nous aurons à les utiliser dans la suite de ce cours.

Création du terrain de jeu pour notre Pong

Dans cette partie, nous allons voir comment concevoir le niveau de notre jeu. Ce que je vous propose c'est d'apprendre en faisant. Comme notre objectif est de créer un clone de Pong, notre scène sera très basique. Nous allons commencer par créer le sol du niveau. Pour cela, cliquez sur le menu « GameObject / 3D Object / Plane » :



Cela va nous permettre de créer une forme plane. Si vous ne touchez à rien, la forme sera sélectionnée par défaut à sa création. Si vous cliquez à côté alors elle sera désélectionnée. Vous pourrez alors cliquer dessus ce plan pour le sélectionner de nouveau. Si un objet est sélectionné vous pourrez visualiser toutes ses propriétés dans l'inspector :

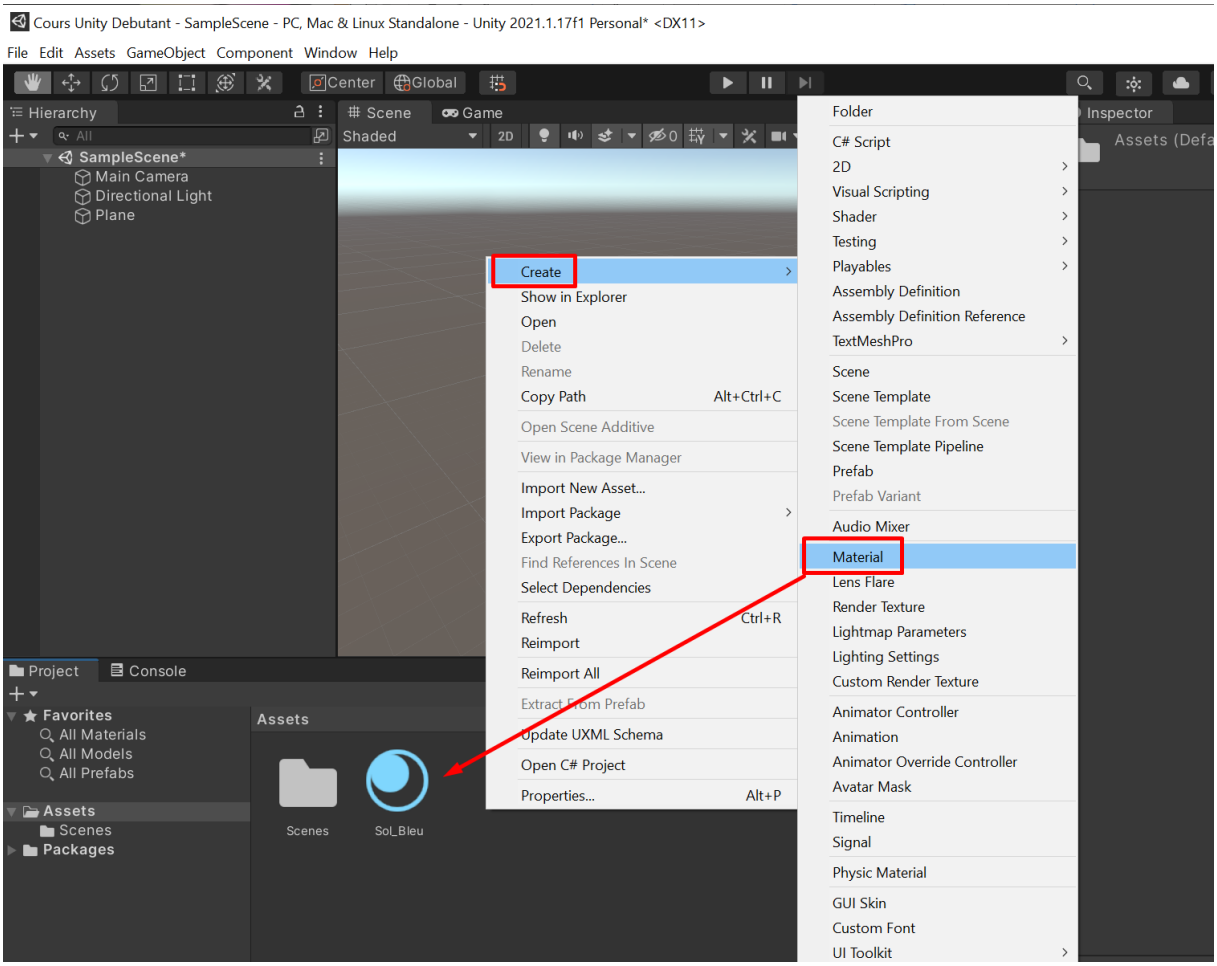


Si on analyse ces différentes propriétés nous pouvons retrouver :

- Transform : correspond à la position, la rotation et l'échelle de la forme 3D.
- Mesh Renderer : correspond au composant qui permet de rendre visible le modèle à l'écran. Si ce composant est décoché (désactivé) alors le modèle 3D sera invisible (mais toujours présent).
- Mesh Collider : c'est le composant qui permet de rendre solide la forme. Grâce au collider, les collisions peuvent être détectées.
- Material : ce composant permet de donner la couleur à l'objet. Un material est un composant complexe qui peut gérer beaucoup de choses comme la couleur, la texture, la façon dont la lumière interagit avec l'objet, la texture de l'objet, la transparence…

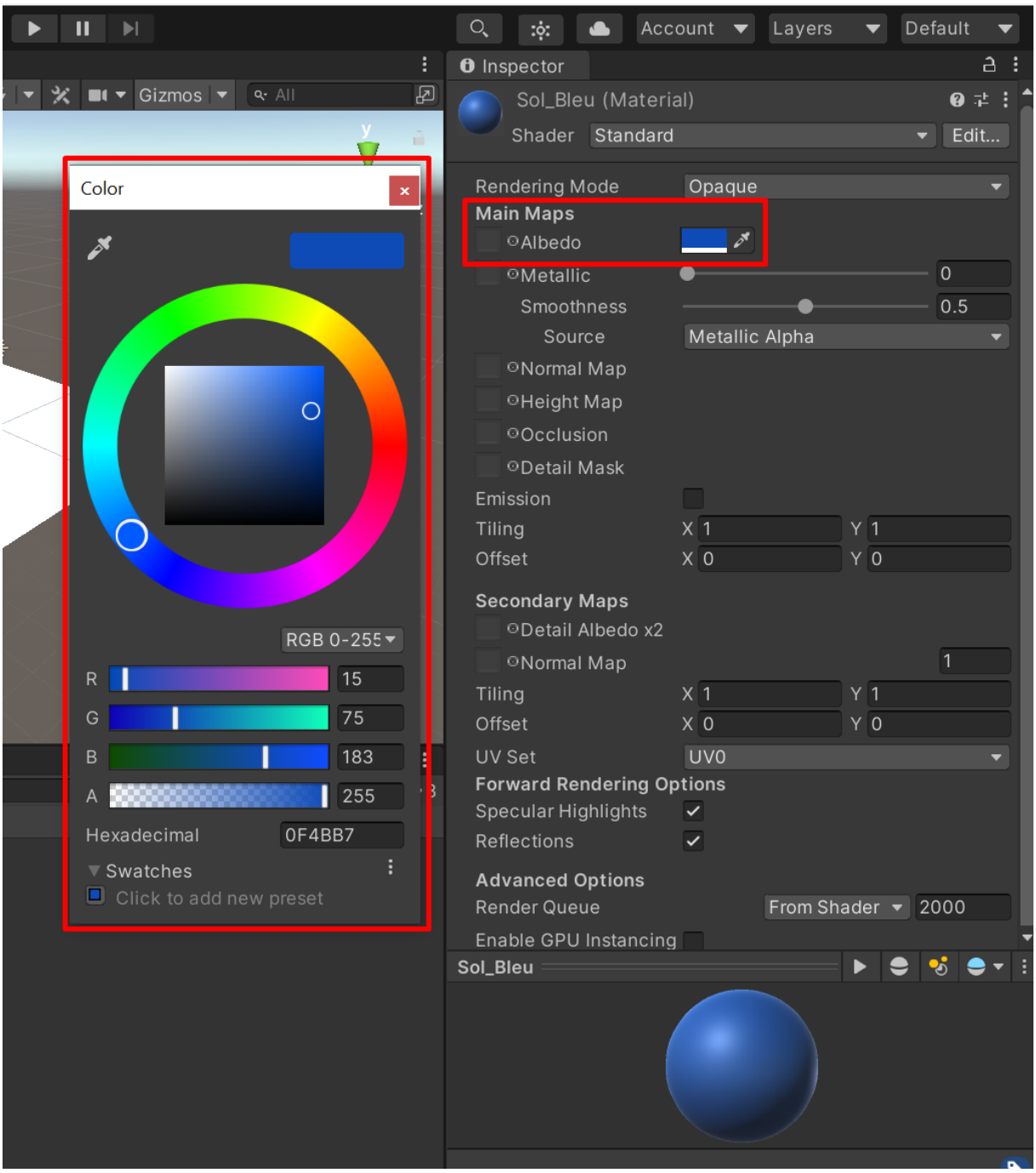
Il est possible de modifier ces propriétés directement via l'inspector ou par glisser/déposer du project sur la scène. Si vous vous souvenez bien, la fenêtre project contient vos ressources. Si vous avez par exemple une texture dans project et que vous la glissez/déposez sur le plan alors le material du plan sera modifié et contiendra la texture déposée.

Dans notre cas nous n'allons pas utiliser de texture. Nous allons plutôt créer un nouveau material avec une couleur spécifique. Dans la fenêtre project faites un clic droit et sélectionnez « Create / Material » :

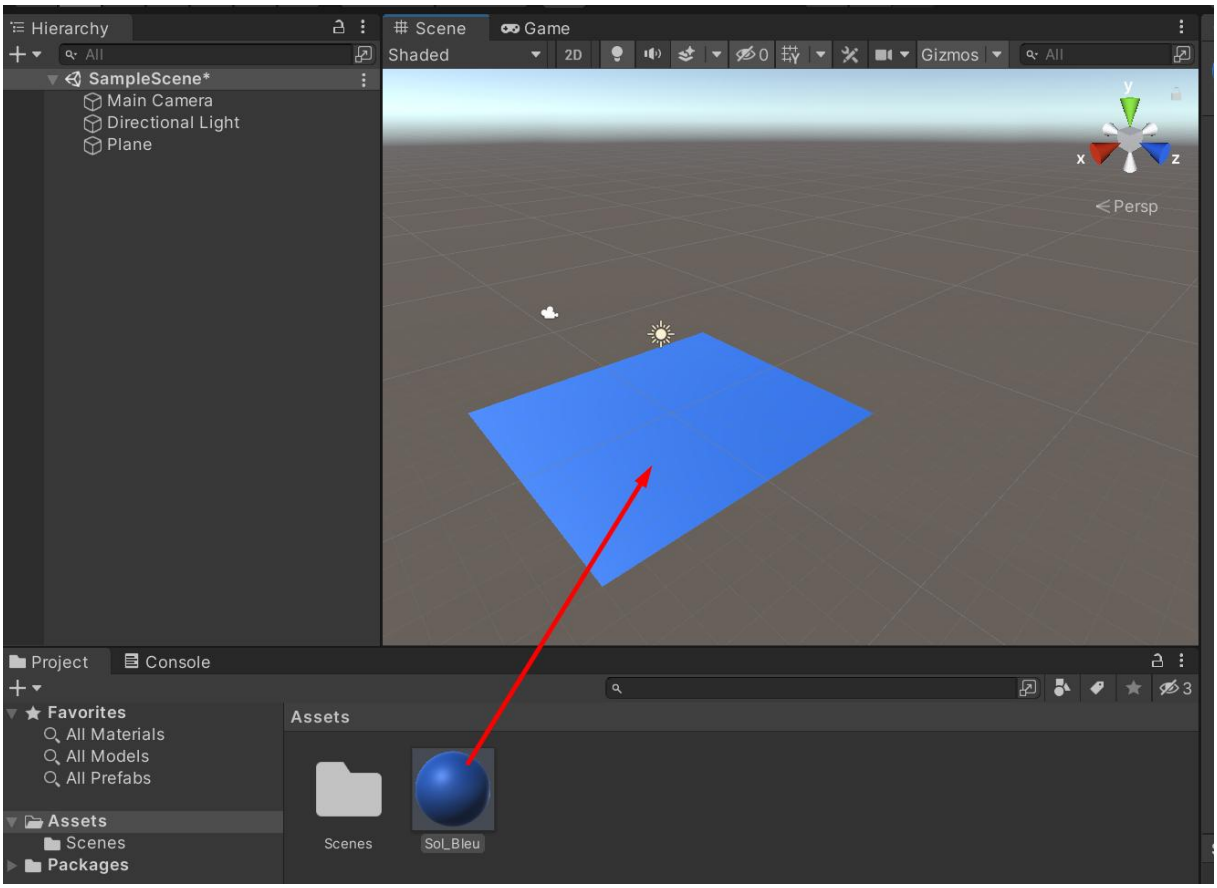


Comme vous pouvez le constater vous pouvez renommer votre material. Dans mon cas j'ai donné un nom au material afin de mieux m'y retrouver.

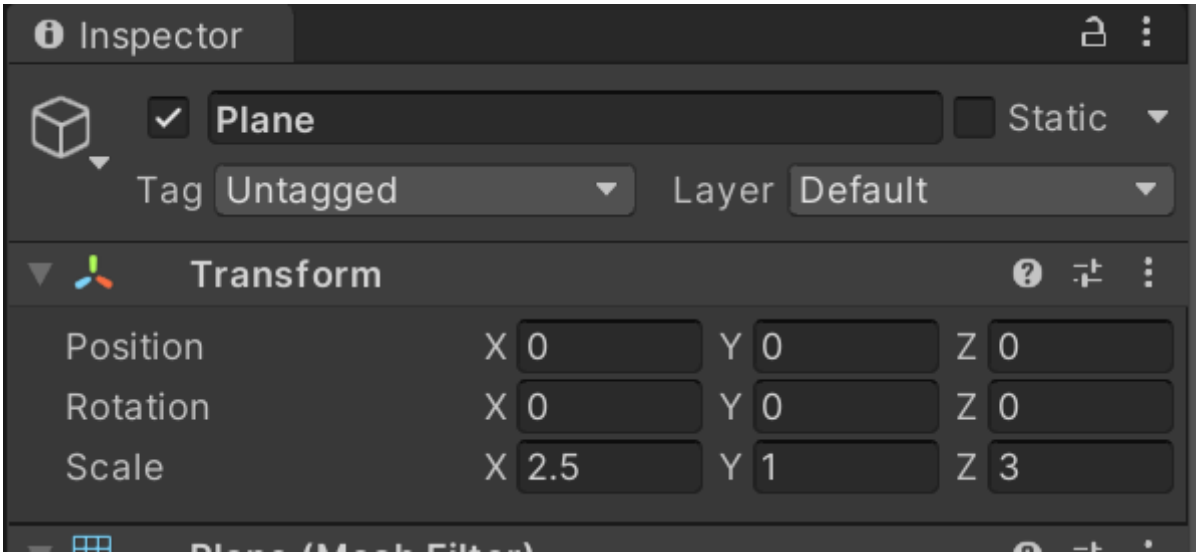
Cliquez sur ce material nouvellement créé afin d'afficher ses propriétés dans l'inspector. Modifiez sa couleur via le color picker afin de choisir une teinte bleu foncé :



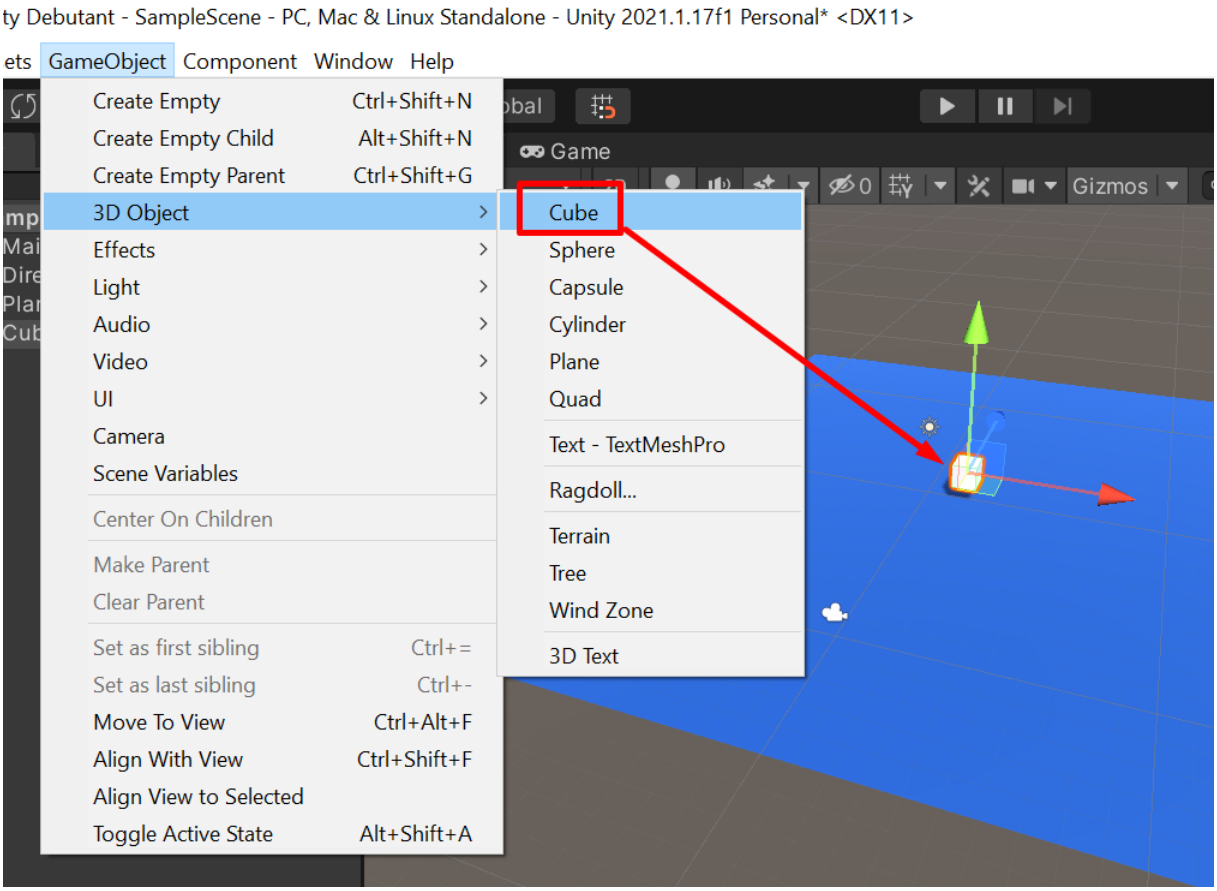
Une fois votre material prêt, faites un glisser/déposer de celui-ci sur le plan de la scène :



Maintenant nous allons modifier la taille du plan afin que tout le monde ait exactement la même forme que moi. Nous allons agrandir le plan et l'étendre sur la longueur. Cliquez sur le plan afin d'afficher ses propriétés dans l'inspector et modifiez la position et le scale en utilisant les mêmes paramètres que moi :



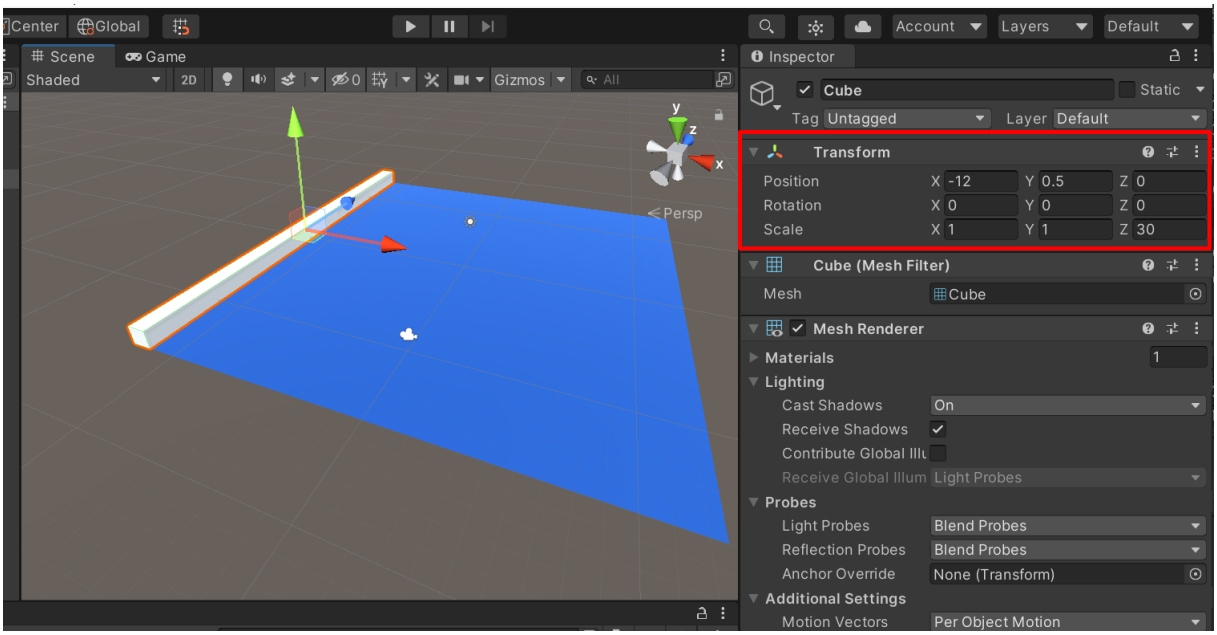
Nous allons maintenant ajouter des bordures à gauche et à droite de notre niveau. Pour cela nous allons créer un cube via le menu « GameObject / 3D Object / Cube » :



Positionnez le cube (via l'inspector) en -12, 0.5, 0.

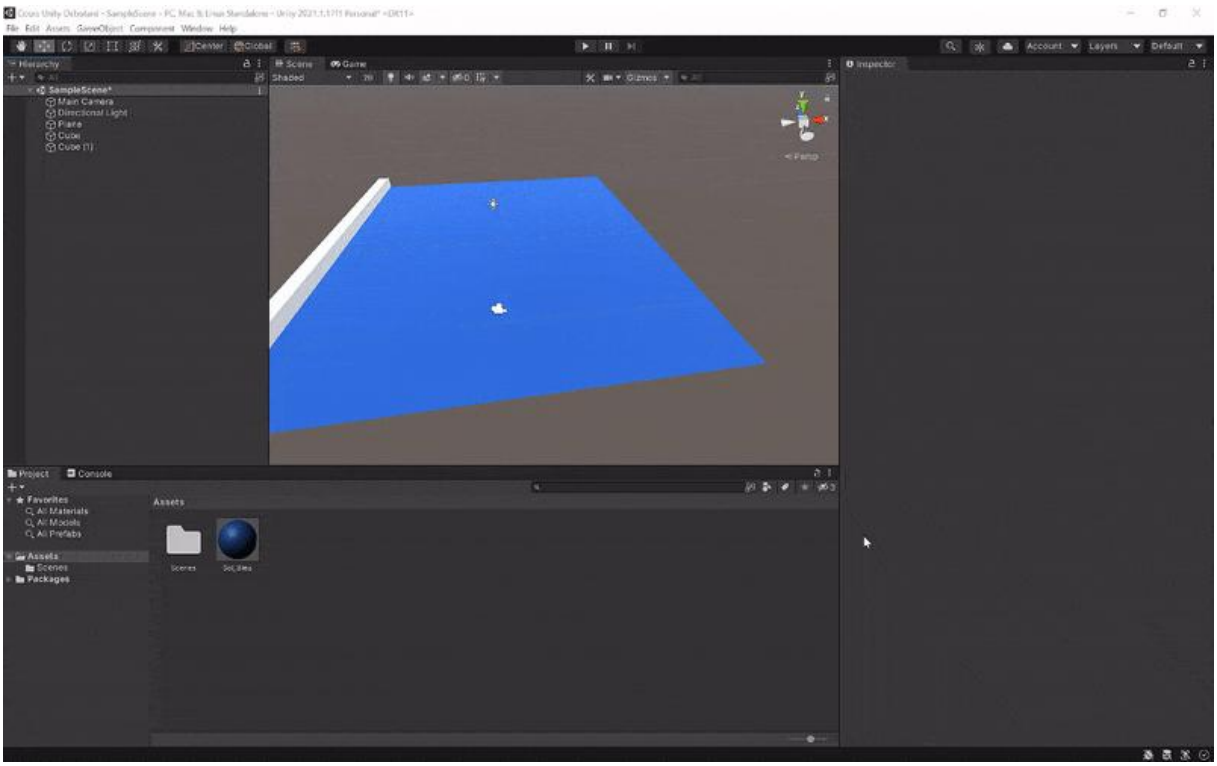
Modifiez sa taille en 1, 1, 30.

Le résultat devrait être le suivant :



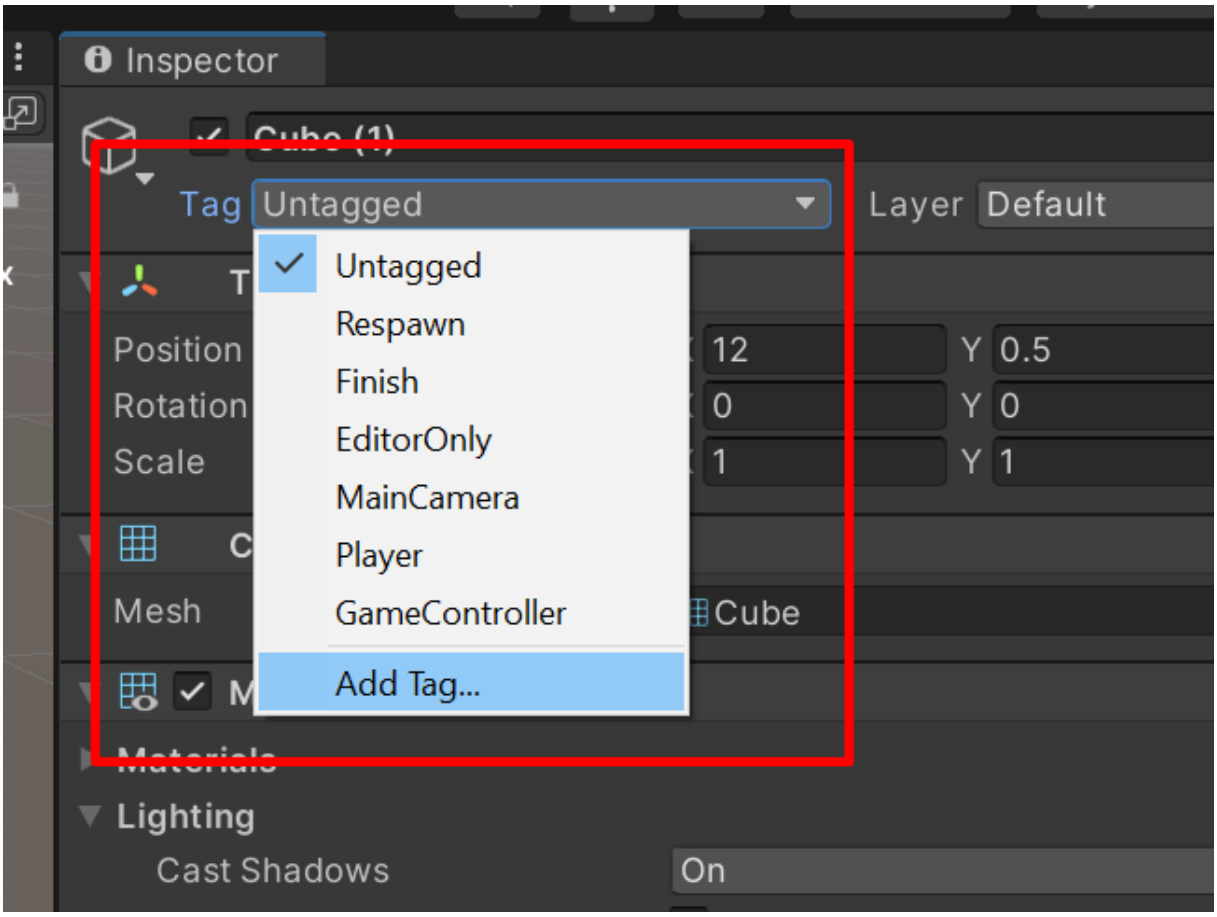
Vous pouvez alors dupliquer ce cube. Pour cela, avec le cube sélectionné, faites CTRL + D. Le cube sera alors dupliqué. Vous ne pouvez pas le voir car les 2 cubes sont superposés mais si vous regardez la fenêtre hierarchy vous verrez les 2 cubes.

Le second cube étant sélectionné, maintenez la touche CTRL de votre clavier et utilisez la flèche rouge (il s'agit d'une flèche visible parmi les 3 axes) afin de déplacer le cube. La touche CTRL permet de déplacer le cube cran par cran afin de garder une position précise. L'idée est de placer le second cube à l'autre extrémité du sol (soit x = 12) :

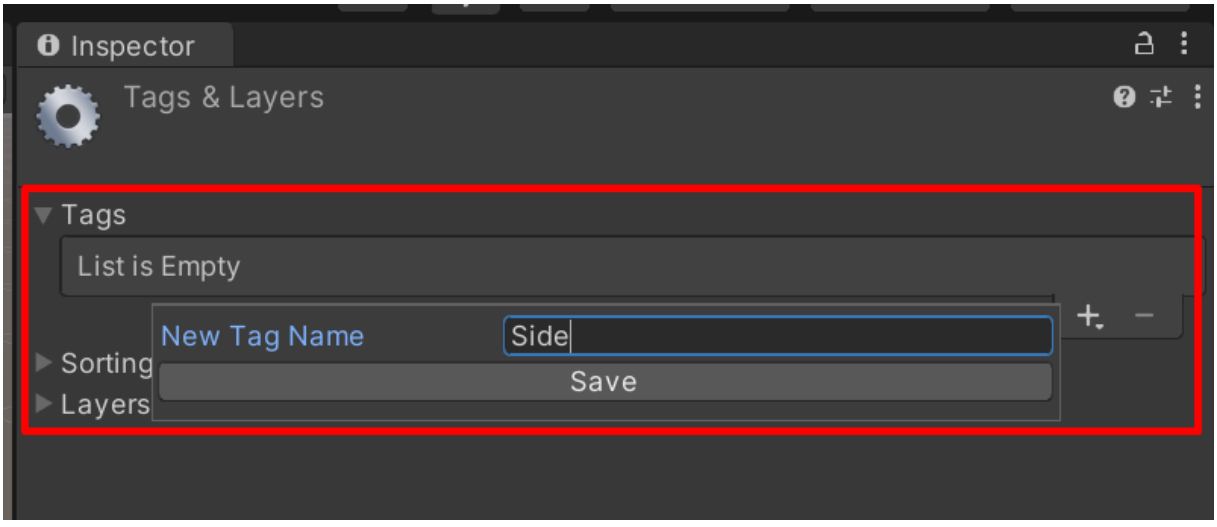


Dans Unity nous avons la possibilité de renommer les objets (on en a déjà parlé, c'est utile pour s'y retrouver dans la fenêtre hierarchy) et nous avons aussi la possibilité d'ajouter un tag à un objet. Le tag permet en général de classer un objet. On peut par exemple imaginer de taguer les pièces en 'coin', les portes en 'door', les pièges en 'trap' etc. Le but est ensuite de pouvoir par la suite, interagir avec ces objets via un script C# en les ciblant grâce à ce tag.

Dans notre cas présent, nous allons avoir besoin de tags. Nous allons avoir besoin de savoir si la balle touche un côté (un mur) du niveau afin de pouvoir faire rebondir la balle. Je vous propose donc de créer le tag 'Side' (qui signifie côté) afin de pouvoir taguer les côtés du niveau. Pour créer un tag, cliquez sur le menu déroulant 'Tag' via l'inspector (attention, pour voir ce menu, un objet doit être sélectionné) et sélectionnez 'Add Tag' :

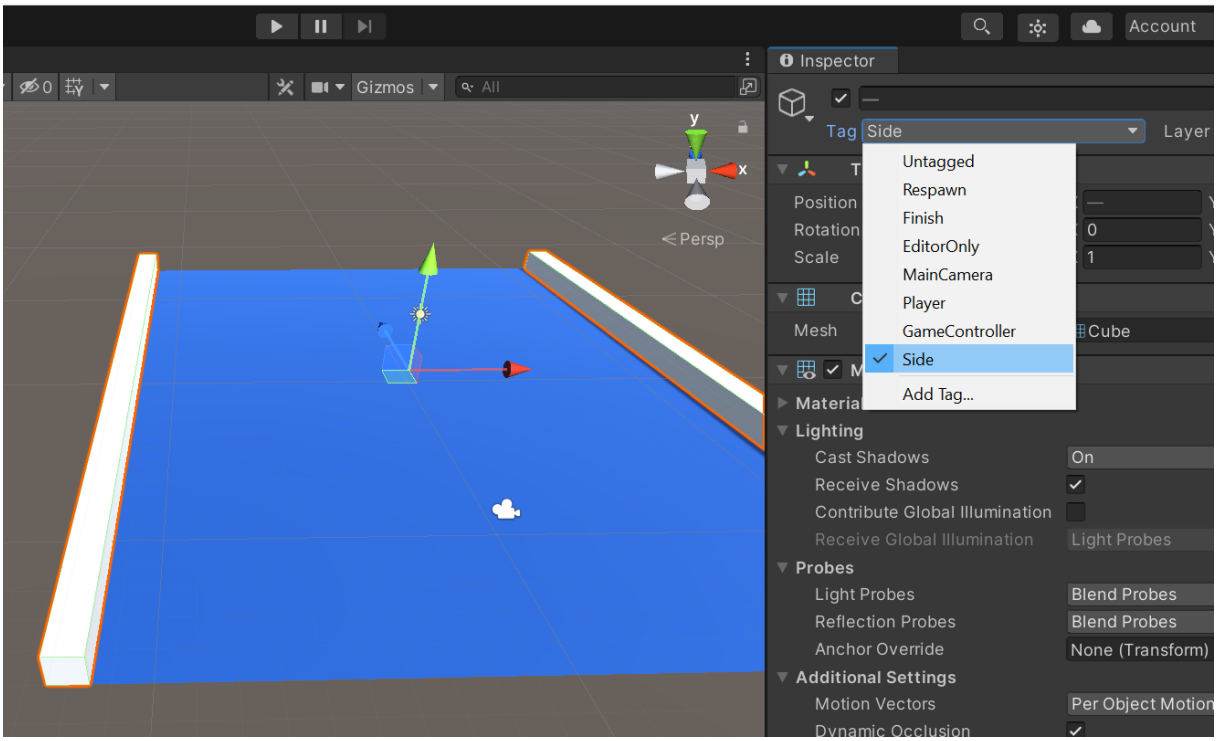


Puis dans la liste des tags, cliquez sur le petit plus pour ajouter un tag personnalisé et saisissez « Side » :



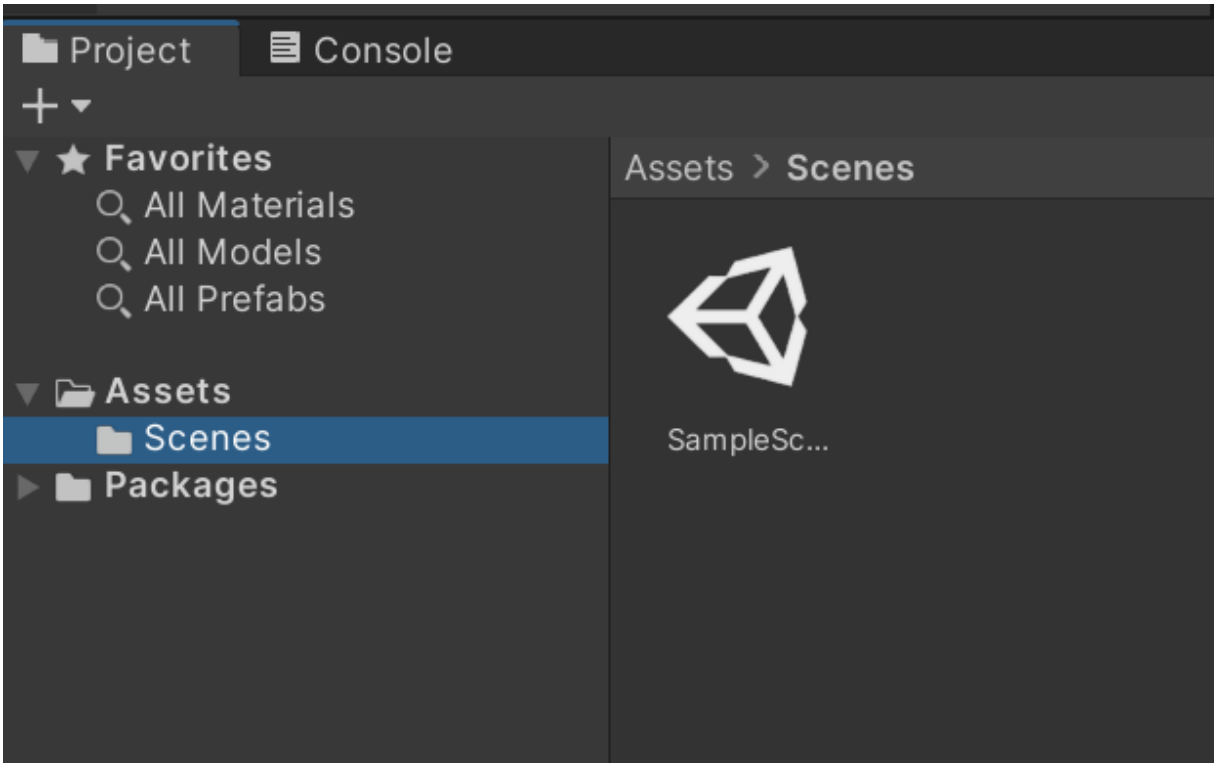
Pensez à cliquer sur « Save ». Retournez sur la scène et sélectionnez le mur de gauche. Maintenez la touche CTRL enfoncée et cliquez sur le mur de droite. Cela permet de faire une sélection simultanée des deux objets.

Avec ces deux objets sélectionnés, retournez dans le menu déroulant des tags afin de sélectionner le tag nouvellement créé pour l'appliquer à la sélection :



Les côtés de notre niveau seront maintenant facilement identifiables dans un script C#. Nous verrons cela dans une prochaine section du cours.

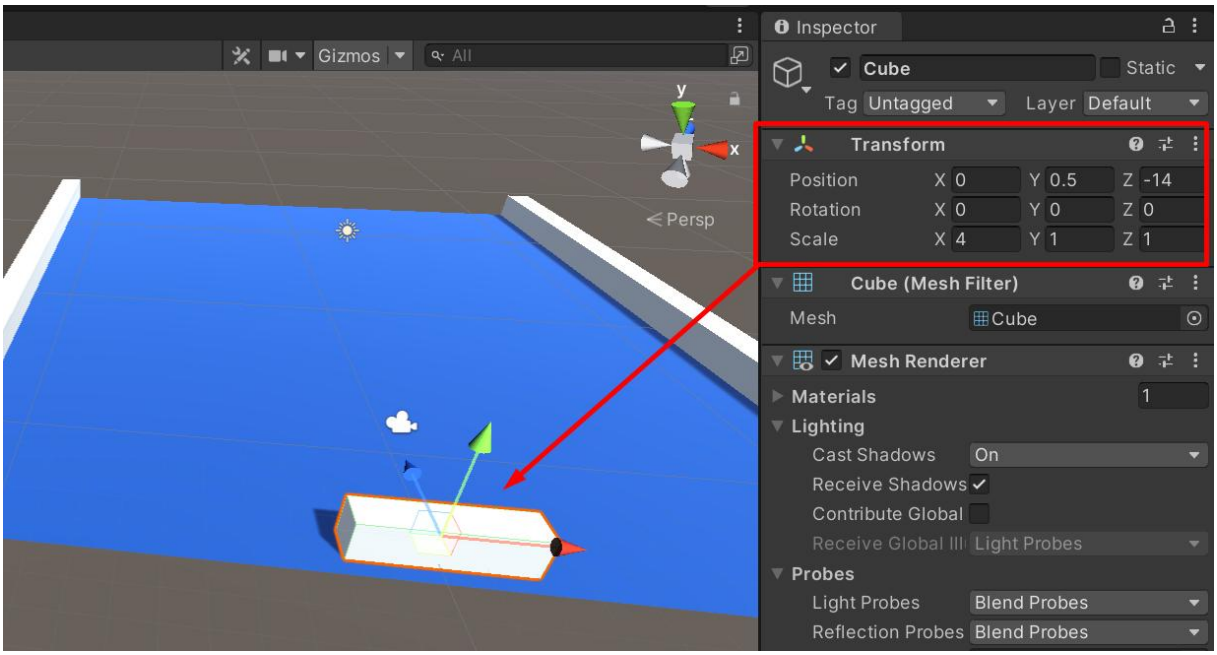
A chaque fin de section, pensez bien à enregistrer votre travail avec un CTRL + S. Cela sauvegardera la scène actuelle qui s'appelle par défaut « SampleScene » et qui se trouve dans le dossier « Scenes » :



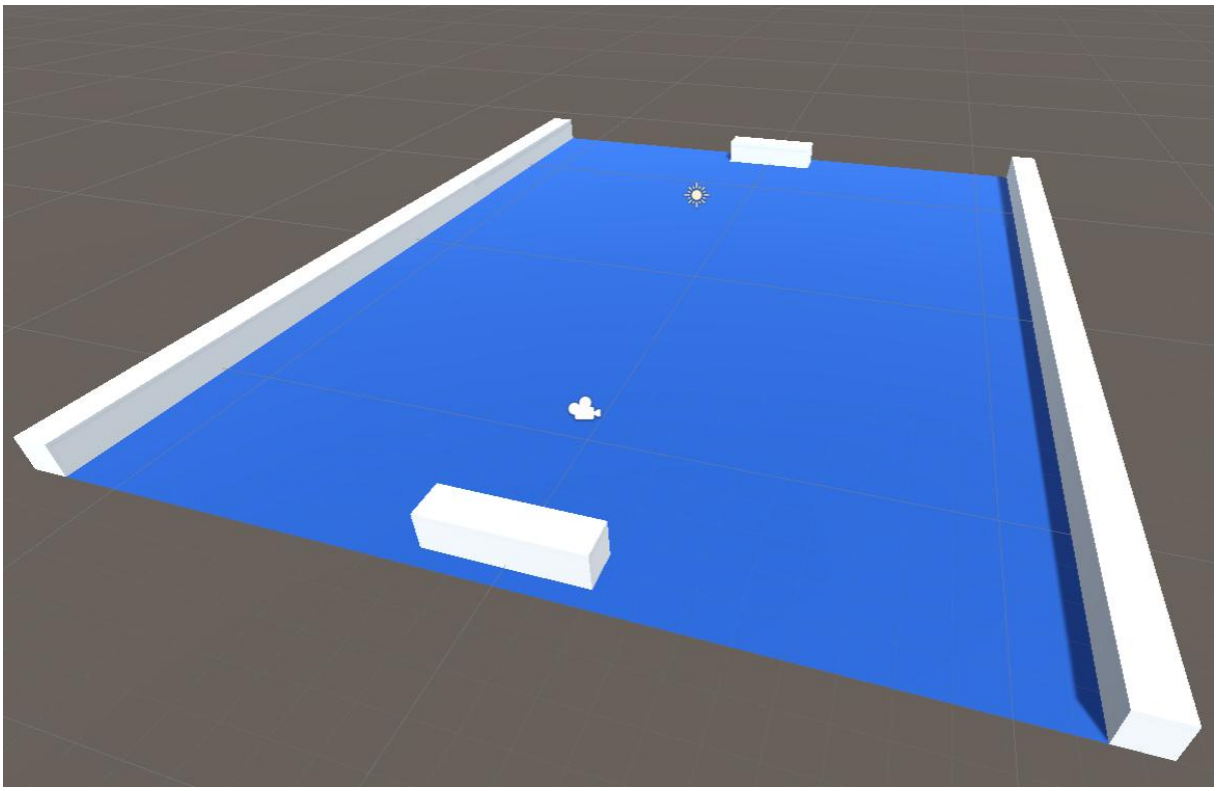
Création des raquettes des joueurs

Ce que j'appelle les raquettes, ce sont les barres qui représentent les joueurs. Ces barres qui permettent de se renvoyer la balle ; le but étant de ne pas laisser passer la balle derrière soi.

Pour créer ces barres, nous allons encore une fois partir d'un simple cube (GameObject / 3D object / Cube) que nous allons étendre sur l'axe X pour lui donner une longueur de 4 unités. La barre contrôlée par le joueur sera positionnée en 0, 0.5, -14 :



Il ne vous reste plus qu'à dupliquer cette barre (CTRL + D) et placer la copie en Z = 14 pour créer la barre contrôlée par l'ordinateur :



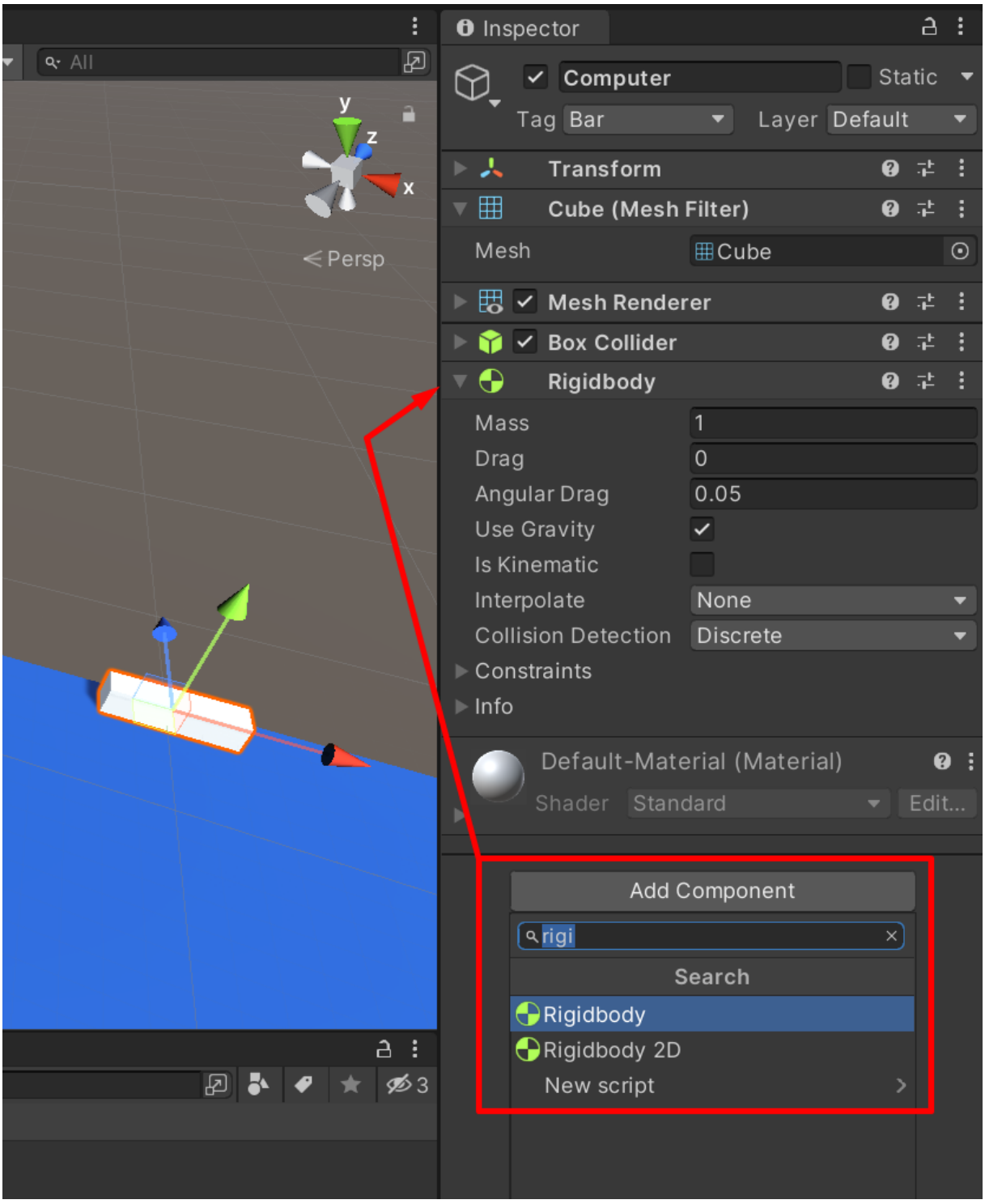
Je vous propose ensuite de créer le tag « Bar » que vous utiliserez pour les 2 barres. Pensez aussi à renommer les barres en « Player » et « Computeur » pour mieux vous y retrouver.

Pour ces 2 barres, nous allons avoir besoin d'ajouter un peu de physique afin de pouvoir détecter des collisions avec la balle. Ce que nous allons faire ici n'est en général pas nécessaire. Cependant, la façon dont la balle sera gérée nécessite de faire la manipulation que je vais vous présenter pour que les collisions soient détectées (en temps normal, seul le collider est nécessaire).

Pour notre cas particulier, nous allons avoir besoin d'ajouter un RigidBody aux barres. Le RigidBody est un composant utilisé en général pour faire en sorte qu'un objet subisse la gravité. Si vous créez un cube et que ce cube possède un RigidBody alors il tombera comme un objet qui tomberait dans le monde réel. Par défaut les objets « flottent », ils ne sont pas affectés par la gravité.

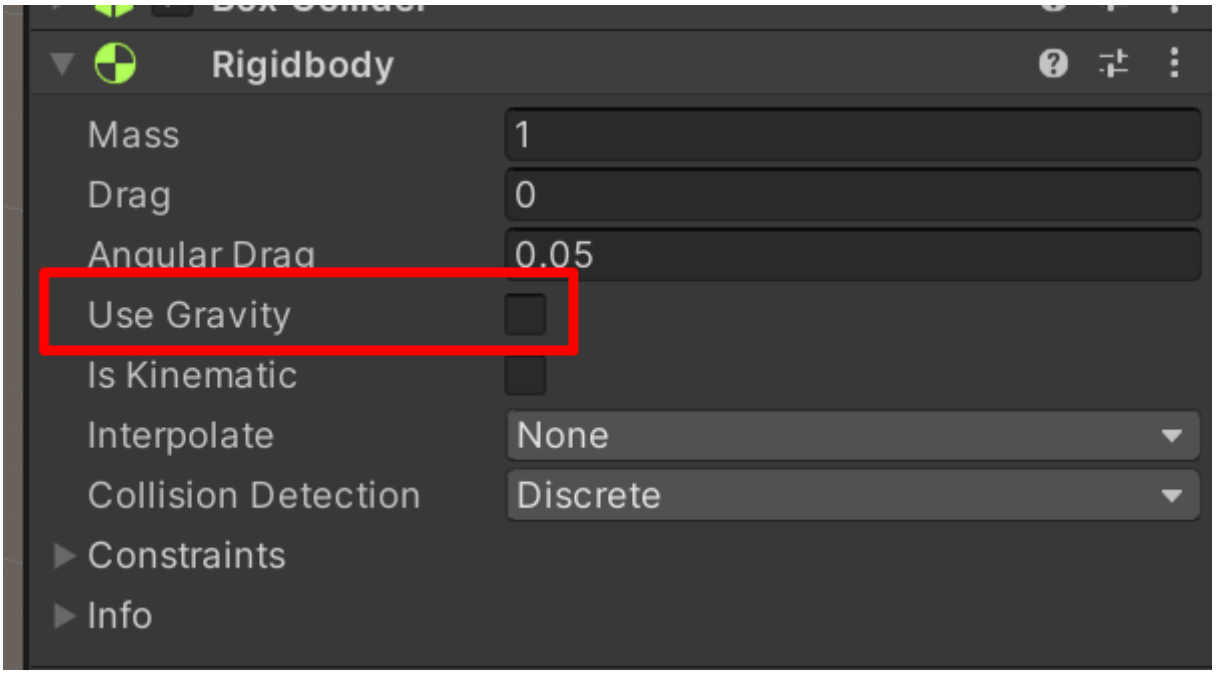
Pour ajouter un RigidBody à un objet, il y a plusieurs façons possibles. Vous pouvez sélectionner un objet (dans notre cas la barre joueur) et utiliser le menu « Component / Physics / Rigidbody ». Le Rigidbody sera alors ajouté à votre objet.

La seconde méthode consiste à sélectionner l'objet (pour ce second exemple prenons la barre contrôlée par l'ordinateur) et à utiliser le bouton « Add Component » visible dans l'inspector. Vous pourrez alors soit naviguer soit saisir le nom du composant que vous recherchez afin de l'ajouter :



Attention de ne pas confondre avec le Rigidbody 2D qui lui est spécifique aux jeux en 2D.

Maintenant, sélectionnez les barres player et computer (en maintenant CTRL pour une sélection multiple). Cela permet de voir les composants des deux barres dans l'inspector. Décochez ensuite la propriété « Use gravity » du Rigidbody :



Cette propriété permet, comme son nom l'indique, de faire en sorte que l'objet soit affecté par la gravité. Dans notre cas ce n'est pas ce que nous voulons. Il n'y a pas de gravité dans notre jeu. Nous avons simplement besoin du Rigidbody pour permettre la détection des collisions entre la barre et la balle.

Plus tard, si nous ne changeons rien, nous rencontrerons un problème avec les barres. En effet, le Rigidbody permet à la physique de s'appliquer sur un objet. En d'autres mots, lorsque la balle viendra percuter la barre, alors la collision provoquera une sorte de force d'impact qui propulsera la barre en arrière. Ce n'est pas ce que nous voulons, la barre doit rester en place et ne pas subir de forces extérieures. Nous allons donc figer la rotation ainsi que la position des barres sur l'axe Z. Pour cela, tout en ayant encore les deux barres sélectionnées, dépliez le sous menu « Constraints » du Rigidbody afin de le paramétrer comme ceci :

De cette façon, les barres ne pourront ni pivoter, ni être déplacées sur un axe autre que l'axe X (gauche / droite).

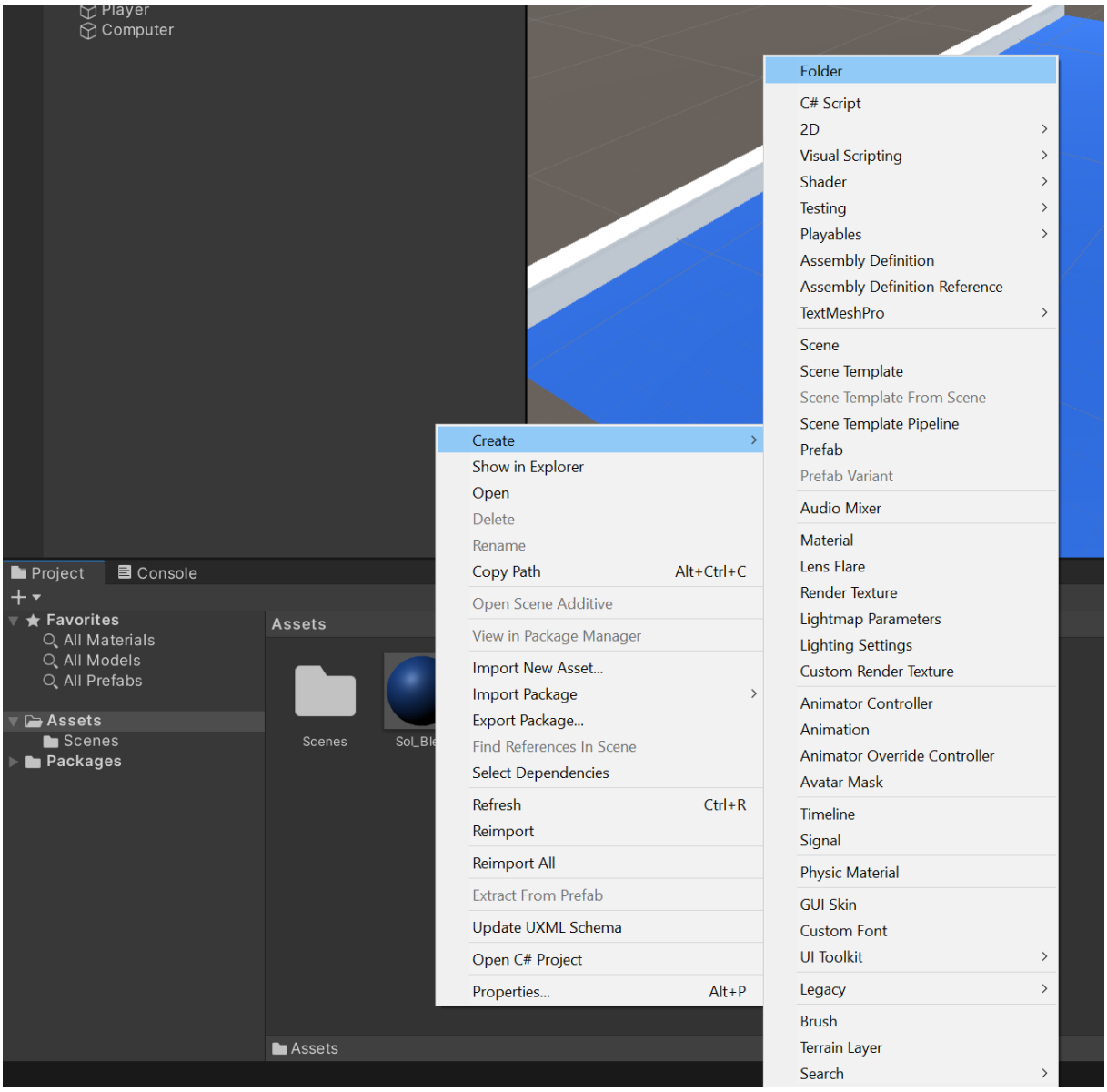
Pour conclure sur cette section, sachez qu'il y a un petit point d'interrogation visible dans l'inspector sur tous les composants qui existent dans Unity. Il s'agit d'un lien vers la documentation qui vous permettra d'en savoir plus sur les propriétés et les fonctionnalités des différents composants de Unity.

Création du script de déplacement de la raquette

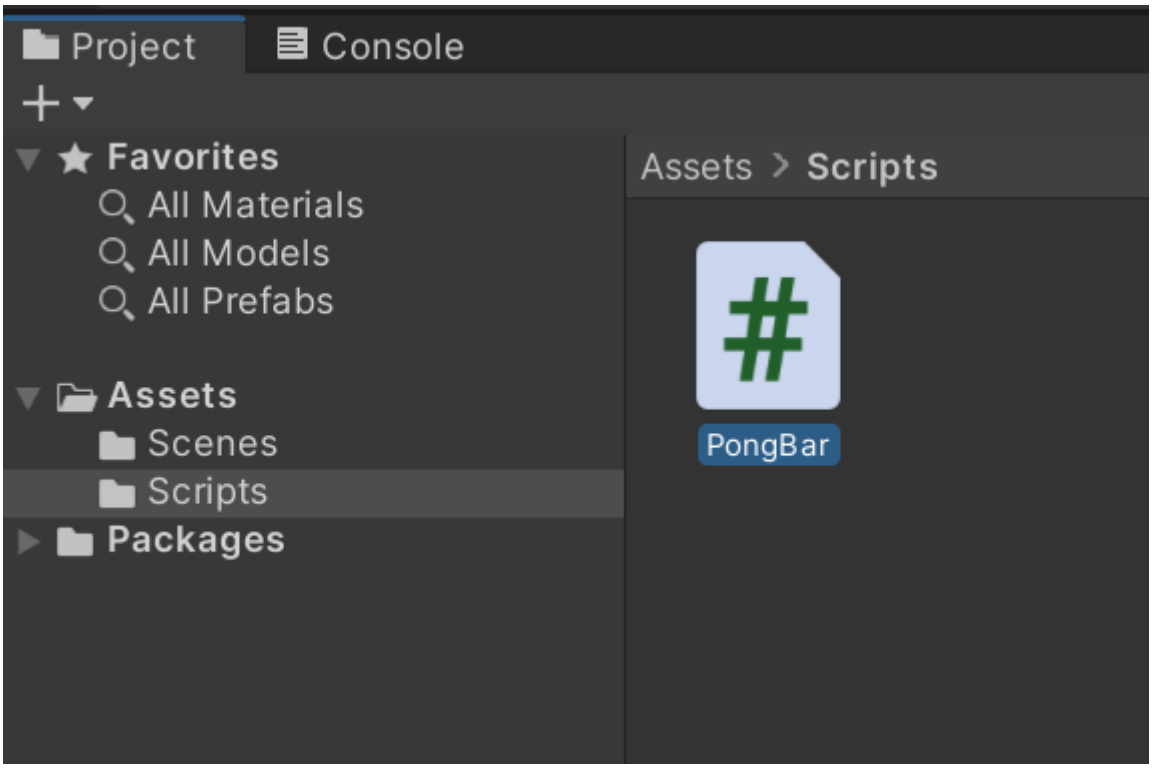
Dans cette section, nous allons créer le script C# des raquettes. Dans Unity, la programmation se fait avec le langage C#. L'écriture des scripts se fait sous Visual Studio (bien qu'il soit possible d'utiliser d'autres éditeurs de code). Les scripts créés sous Visual Studio peuvent ensuite être utilisés dans Unity. Pour cela, il faudra faire un glisser/déposer du script sur l'objet sur lequel on souhaite attacher le script. Un script peut être réutilisé plusieurs fois sur plusieurs objets différents. Dans notre cas nous aurons le même script pour les deux raquettes.

Des notions en programmation sont nécessaires pour bien comprendre cette section. J'essayerai cependant de tout détailler. Si vous ne connaissez pas C# mais que vous avez des notions dans un autre langage orienté objet alors c'est parfait pour comprendre les exemples présentés ici.

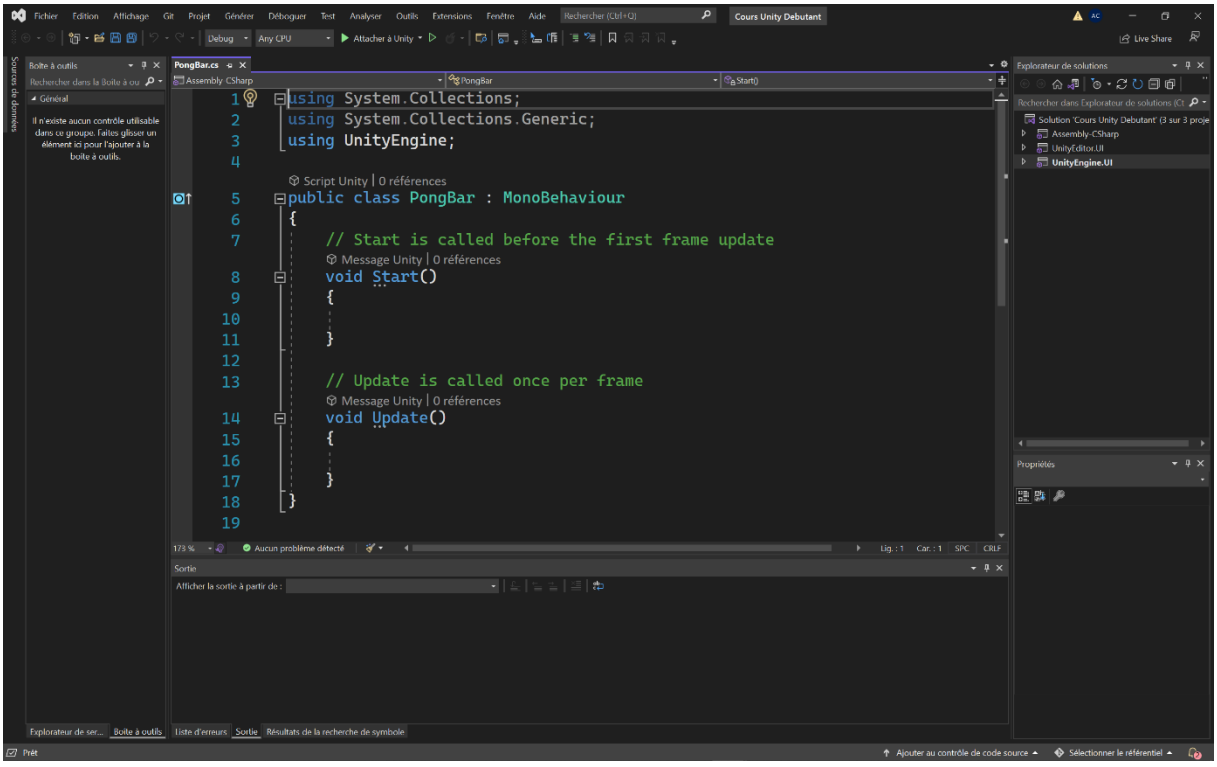
Avant de créer un script, je vous invite à créer un nouveau dossier dans la fenêtre project afin d'y stocker les scripts. Pour créer un dossier, cliquez sur le dossier « Assets » puis faites un clic droit dans la fenêtre project et cliquez sur « Create / Folder » :



Appelez ce dossier « Scripts » et ouvrez-le. Dans ce dossier, faites un clic droit puis choisissez « Create / C# Script » afin de créer un script que vous appellerez « PongBar » :



Double cliquez sur ce script afin de l'ouvrir dans Visual Studio :



Vous constatez que le script n'est pas vide. Si nous analysons le contenu du script il y a dans l'ordre :

- Des « using » qui permettent d'inclure des fonctionnalités comme UnityEngine qui permet d'utiliser les fonctions de Unity dans le code C#.

- La classe « PongBar » dont le nom correspond au nom du script. Il est d'ailleurs indispensable que le script et la classe aient exactement le même nom sinon cela ne fonctionnera pas.
- La fonction Start qui se déclenche au lancement du programme.
- La fonction Update qui tourne en boucle à l'infini. Si votre jeu tourne en 60 images par seconde alors Update s'exécutera 60 fois par seconde.

Voilà en quelques lignes le descriptif du code par défaut. Nous n'allons pas avoir besoin de la fonction Start. Vous pouvez donc la supprimer.

Dans notre script, nous allons avoir besoin de savoir si la raquette est celle du joueur ou celle de l'ordinateur. Si vous vous souvenez bien, j'ai dit que nous allons utiliser ce script pour les 2 raquettes, nous devons donc les différencier.

Nous allons donc créer une variable booléenne (donc la valeur de cette variable ne peut être égale qu'à vrai ou faux). Cette variable permettra de savoir si le script est attaché au joueur humain :

```
public bool isHumanPlayer = false;
```

Comme cette variable est définie comme « public », la valeur sera visible et modifiable dans l'inspector. Cela nous permettra d'indiquer la valeur selon s'il s'agit de la raquette player ou computer.

La raquette aura également une vitesse de déplacement. Nous allons créer une autre variable de type float cette fois-ci (nombre à virgule) pour y stocker la vitesse :

```
public float speed = 15;
```

Maintenant, à l'intérieur de la fonction Update, créez une variable qui stockera le mouvement de la raquette :

```
float move;
```

La barre pourra être déplacée avec les flèches gauche et droite du clavier. Pour détecter l'appuie sur une touche il faut utiliser le mot-clé Input. Dans la classe Input nous avons accès à toutes les touches. Parmi ces touches, nous retrouvons les axes (horizontal / vertical). Cela nous permet de tester les flèches du clavier. Pour récupérer la valeur de l'axe horizontal il faudra donc utiliser :

```
Input.GetAxis("Horizontal")
```

Nous allons donc utiliser cela pour définir la valeur de notre variable move. Il faudra également multiplier cette valeur par speed (notre variable vitesse) afin que la barre se déplace à la vitesse donnée. Enfin nous devront multiplier également par Time.deltaTime, ce qui nous permettra d'avoir la même vitesse de déplacement peu importe la puissance de l'ordinateur. Cela nous donne donc le bout de code suivant :

```
move = Input.GetAxis("Horizontal") * speed * Time.deltaTime;
```

Ce déplacement ne doit se faire que s'il s'agit du joueur. Nous devons donc entourer cette ligne de la condition suivante :

```
if (isHumanPlayer)

{

move = Input.GetAxis("Horizontal") * speed * Time.deltaTime;

}
```

Si vous souhaitez également contrôler la raquette de l'ordinateur (par exemple pour faire des tests durant le développement), vous pouvez ajouter ce bout de code :

```
else

{

move = Input.GetAxis("Vertical") * speed * Time.deltaTime;

}
```

Tout ceci nous permet de stocker le mouvement souhaité dans la variable move. Mais ce mouvement n'est pas appliqué, il est juste stocké. Pour l'appliquer, il faudra utiliser la fonction Translate (translation) sur le Transform (si vous vous rappelez bien, le transform correspond à la position, rotation et taille de l'objet) afin de matérialiser le mouvement.

Notre variable move est un float. La fonction Translate attend un Vector3 en entrée (vecteur à 3 dimensions X, Y et Z). Nous devons donc multiplier move par un Vector3 afin que le type corresponde à ce qui est attendu :

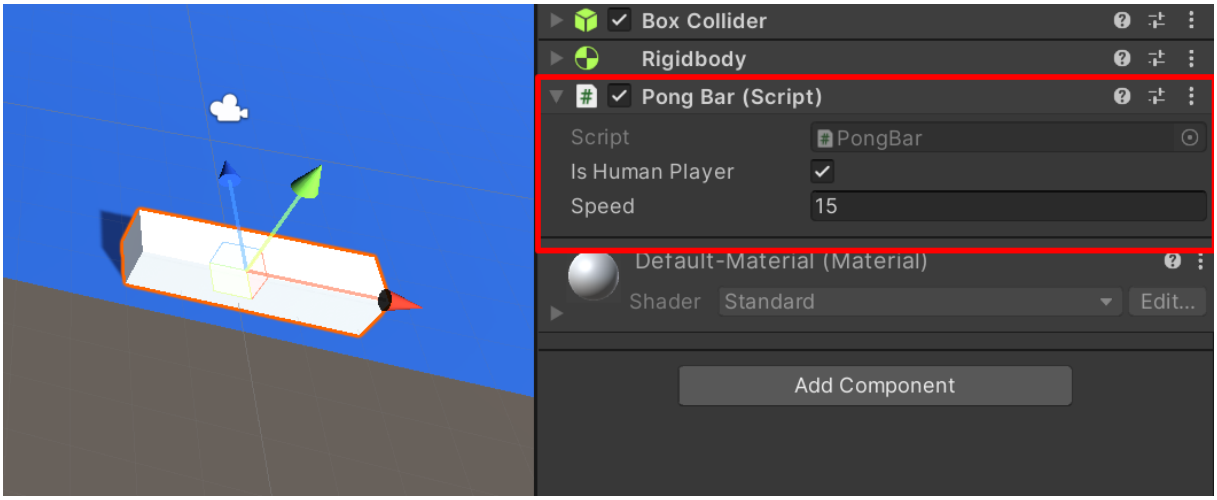
```
transform.Translate(move * Vector3.right);
```

Vous devriez avoir le code suivant :

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PongBar : MonoBehaviour
{
   public bool isHumanPlayer = false;
   public float speed = 15;

   void Update()
   {
      float move;

      if (isHumanPlayer)
      {
         move = Input.GetAxis("Horizontal") * speed * Time.deltaTime;
      }
      else
      {
         move = Input.GetAxis("Vertical") * speed * Time.deltaTime;
      }

      transform.Translate(move * Vector3.right);
   }
}
```

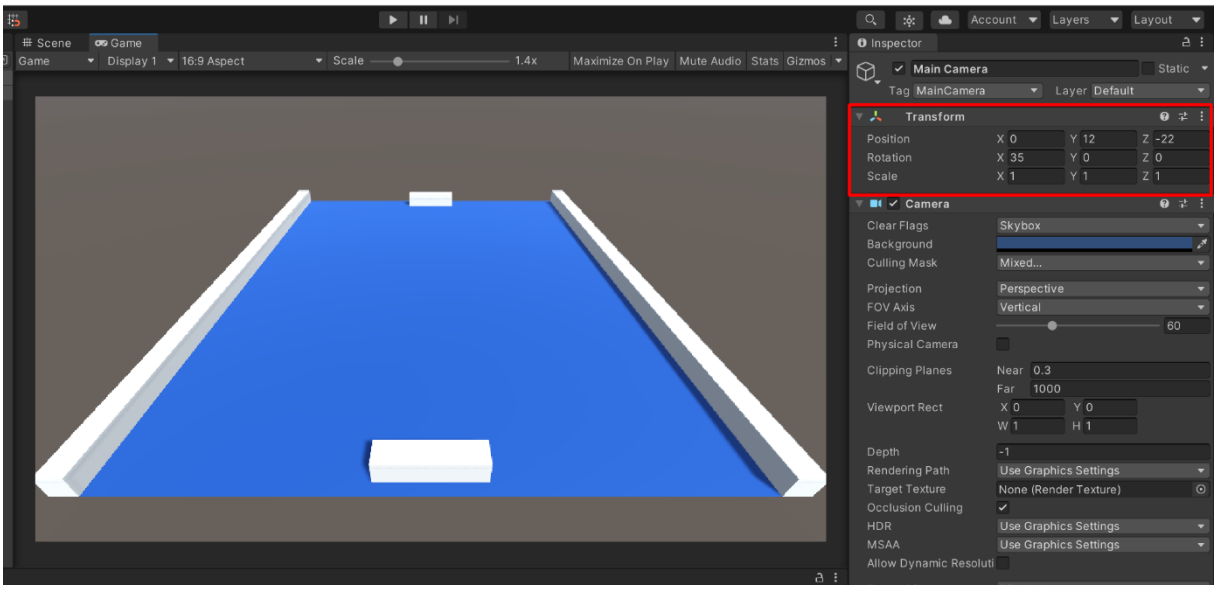
Afin de tester ce script, enregistrez-le. Retournez dans Unity et vérifiez dans la console si vous avez une erreur. Si tel est le cas, corrigez-la. Ensuite, ajoutez ce script aux 2

raquettes avec un glisser/déposer du script sur l'objet. Vérifiez via l'inspector que le script est bien ajouté :

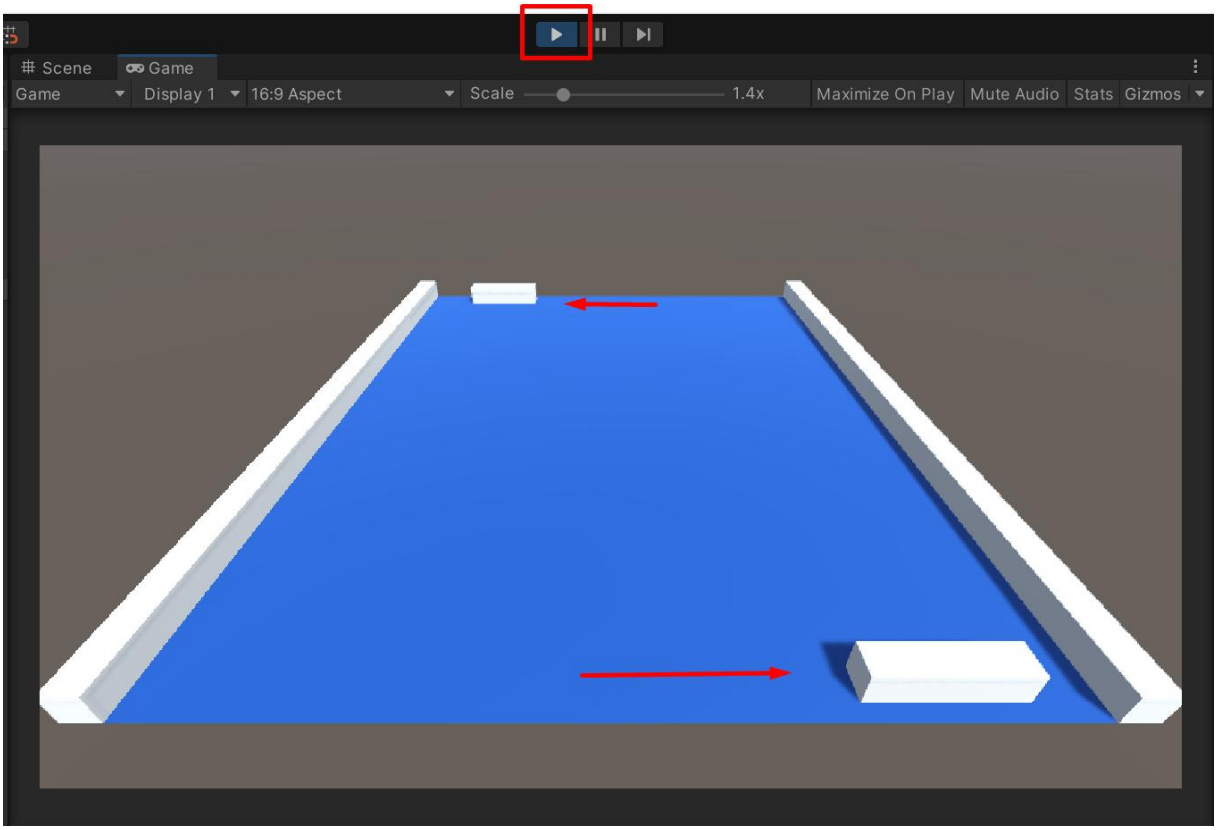


Pensez à cocher la variable « Is human player » sur le script attaché à la raquette du joueur.

Avant de tester notre script, nous allons positionner la caméra comme il faut. Pour cela, sélectionnez la caméra et ajustez les valeurs du transform comme ceci :



Cliquez ensuite sur l'icône « Play » en haut de l'écran pour lancer le jeu. Utilisez les flèches du clavier afin de vérifier que la barre se déplace convenablement :



Si tout fonctionne, bravo, vous avez écrit votre premier bout de code fonctionnel. Vous pouvez cliquer sur Play de nouveau pour sortir du jeu.

Notre script n'est cependant pas parfait. En effet, les raquettes peuvent se déplacer à l'infini, dépasser les murs et sortir de l'écran. Pour remédier à cela, retournez dans le script et ajoutez la variable suivante :

```
private float xMaxDistance = 9.5f;
```

Cette valeur correspond à la distance maximale autorisée sur l'axe X.

Ajoutez ensuite deux conditions dans la fonction Update afin de bloquer le déplacement lorsque cette distance maximale est atteinte :

```
if (transform.position.x < -xMaxDistance)

{

transform.position = new Vector3(-xMaxDistance, transform.position.y,
transform.position.z);

}


if (transform.position.x > xMaxDistance)
```

```
{
transform.position = new Vector3(xMaxDistance, transform.position.y,
transform.position.z);
}
```

Comme vous pouvez le voir, si la position de la raquette sur l'axe X est hors limite, alors je modifie la position de la raquette afin de lui donner sa position actuelle en Y et en Z et je spécifie la valeur de X manuellement.

Le code complet pour ce script est le suivant :

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PongBar : MonoBehaviour
{
  public bool isHumanPlayer = false;
  public float speed = 15;

  private float xMaxDistance = 9.5f;

  void Update()
  {
    float move;

    if (isHumanPlayer)
    {
      move = Input.GetAxis("Horizontal") * speed * Time.deltaTime;
    }
```

```
        else
        {
            move = Input.GetAxis("Vertical") * speed * Time.deltaTime;
        }

        transform.Translate(move * Vector3.right);

        if (transform.position.x < -xMaxDistance)
        {
            transform.position = new Vector3(-xMaxDistance, transform.position.y,
transform.position.z);
        }

        if (transform.position.x > xMaxDistance)
        {
            transform.position = new Vector3(xMaxDistance, transform.position.y,
transform.position.z);
        }
    }
}
```

Vous pouvez maintenant tester de nouveau votre projet et constater que les raquettes ne peuvent plus sortir du terrain de jeu.

Pensez comme d'habitude à enregistrer votre travail avant de passer à la prochaine section.

Code source

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

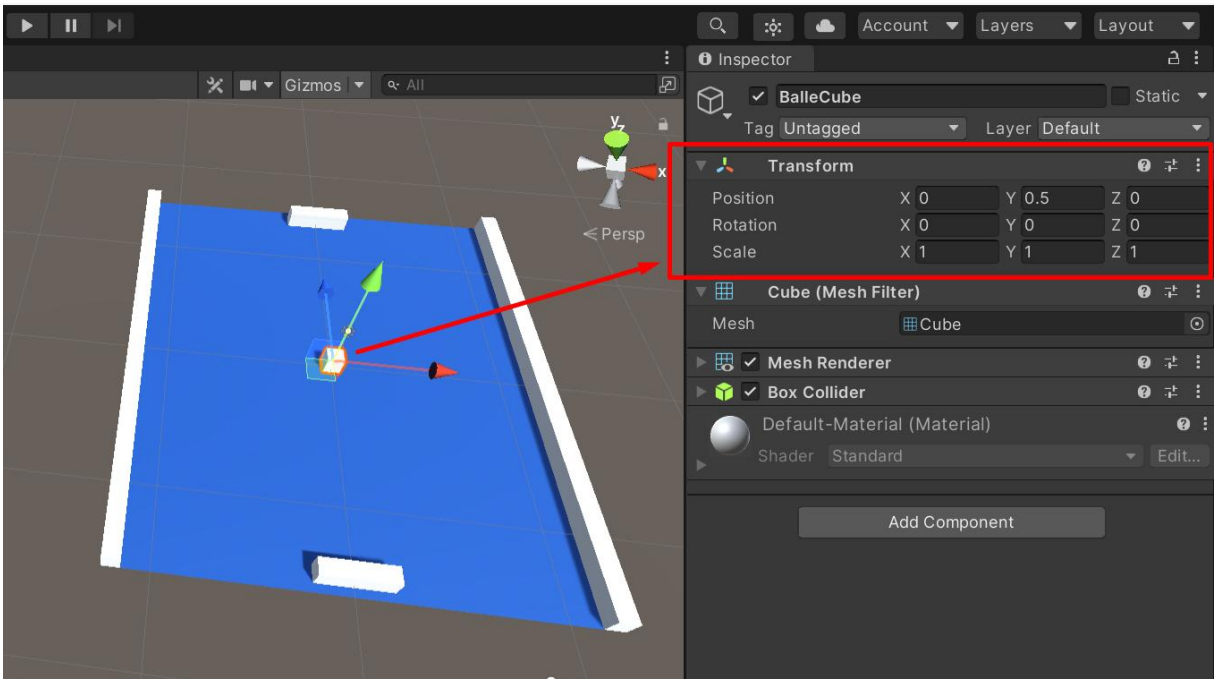
```
public class PongBar : MonoBehaviour
{
   public bool isHumanPlayer = false;
   public float speed = 15;

   private float xMaxDistance = 9.5f;

   void Update()
   {
      float move;

      if (isHumanPlayer)
      {
         move = Input.GetAxis("Horizontal") * speed * Time.deltaTime;
      }
      else
      {
         move = Input.GetAxis("Vertical") * speed * Time.deltaTime;
      }

      transform.Translate(move * Vector3.right);

      if (transform.position.x < -xMaxDistance)
      {
         transform.position = new Vector3(-xMaxDistance, transform.position.y,
transform.position.z);
      }
```

```
    if (transform.position.x > xMaxDistance)
    {
        transform.position = new Vector3(xMaxDistance, transform.position.y,
transform.position.z);
    }
  }
}
```

Création de la balle et de son script

Maintenant que nos raquettes sont fonctionnelles, ce serait bien de pouvoir se renvoyer la balle. Le seul problème c'est que nous n'avons pas de balle !

Créez donc un simple cube que vous placerez en 0, 0.5, 0 (soit au centre du plateau) :



Ajoutez un RigidBody à cette balle et configurer-le de cette façon :

Créez ensuite un script « PongBall » que vous ajouterez également à la balle. Une fois ajouté, ouvrez ce script pour pouvoir l'éditer dans Visual Studio.

Dans ce script nous allons avoir besoin de quelques variables. Il nous faut la vitesse de déplacement qui sera un float, la direction du mouvement qui sera un Vector3 et une distance max sur l'axe Z qui nous permettra de savoir si la balle passe derrière un joueur :

```
public float speed;

public Vector3 direction;

private float zMaxDistance = 15f;
```

Dans la fonction Start nous allons appeler la fonction SetDirection :

```
SetDirection();
```

Cette fonction n'existe pas encore. Nous allons la créer dans un instant. Elle permettra de donner une direction initiale à la balle. Cette fonction sera aussi utilisée pour réinitialiser la balle quand elle passe derrière un joueur. L'intérêt de faire une fonction c'est de pouvoir réutiliser le code à plusieurs endroits du script.

Cette fonction contiendra deux instructions :

- Réinitialiser la position de la balle afin de la placer au centre.
- Donner un mouvement initial aléatoire (Random) et normalisé (Normalize).

Normaliser un mouvement permet d'avoir une vitesse constante. Voilà le code de cette fonction :

```
public void SetDirection()
{
transform.position = new Vector3(0, .5f, 0);
direction = new Vector3(Random.Range(0.75f, 1.75f), 0, -1).normalized;
}
```

Dans la fonction Update, nous allons utiliser la fonction Translate (fonction existante dans Unity et déjà utilisée dans le script de la raquette) afin d'appliquer le mouvement à la balle :

```
transform.Translate(direction * speed * Time.deltaTime);
```

Nous devons également tester avec des conditions si la balle passe derrière le joueur ou derrière l'ordinateur. Si tel est le cas, on réinitialise la balle en utilisant notre fonction :

```
// IA gange
if (transform.position.z < -zMaxDistance && direction.z < 0)
{
SetDirection();
}

// Joueur gange
if (transform.position.z > zMaxDistance && direction.z > 0)
{
SetDirection();
}
```

Dans ces deux tests, je vérifie si la balle passe derrière un des joueurs et que celle-ci continue d'aller derrière le joueur.

Il faut maintenant gérer les rebonds de la balle. Si la balle rentre en collision avec un mur, elle doit rebondir sur l'axe X. Si elle rentre en collision avec une raquette, elle doit rebondir en Z. Pour coder le rebond, il faudra juste multiplier la valeur de l'axe par -1 afin d'inverser la direction.

La particularité du rebond sur la raquette est qu'il faut savoir s'il s'agit de la raquette du joueur ou de l'ordinateur afin de pouvoir appliquer le bon rebond selon la situation. Il est possible depuis le script de la balle d'accéder au script de la raquette afin de récupérer cette info.

Enfin, pour détecter une collision, on peut utiliser la fonction OnCollisionEnter de Unity comme cela :

```
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "Bar")
    {
        bool isPlayer = collision.gameObject.GetComponent<PongBar>().isHumanPlayer;
        if ((isPlayer && direction.z < 0) || (!isPlayer && direction.z > 0))
        {
            direction.z *= -1;
        }
    }

    if (collision.gameObject.tag == "Side")
    {
        direction.x *= -1;
    }
}
```

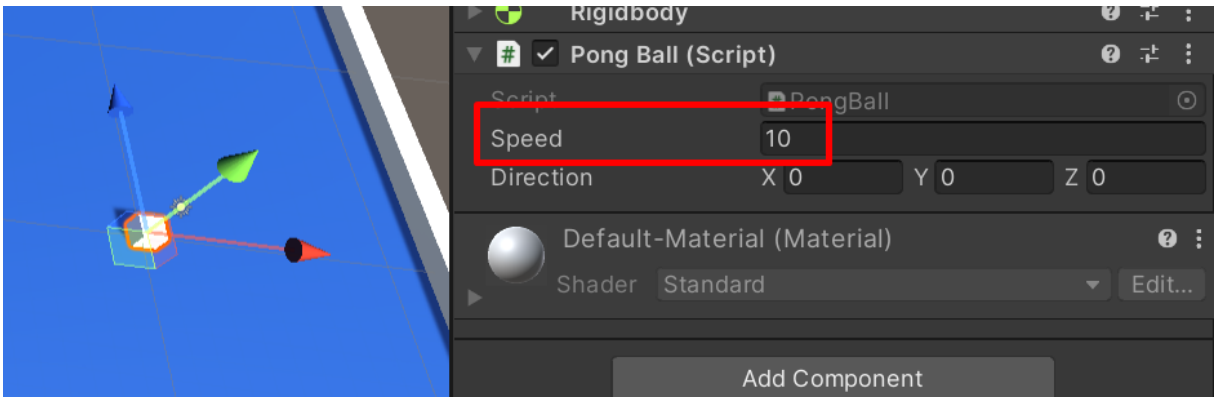
Le script complet est le suivant :

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PongBall : MonoBehaviour
{
   public float speed;
   public Vector3 direction;
   private float zMaxDistance = 15f;

   private void Start()
   {
      SetDirection();
   }

   void Update()
   {
      transform.Translate(direction * speed * Time.deltaTime);

      // IA gange
      if (transform.position.z < -zMaxDistance && direction.z < 0)
      {
         SetDirection();
      }

      // Joueur gange
```

```
    if (transform.position.z > zMaxDistance && direction.z > 0)
    {
        SetDirection();
    }
}

public void SetDirection()
{
    transform.position = new Vector3(0, .5f, 0);
    direction = new Vector3(Random.Range(0.75f, 1.75f), 0, -1).normalized;
}

private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "Bar")
    {
        bool isPlayer = collision.gameObject.GetComponent<PongBar>().isHumanPlayer;
        if ((isPlayer && direction.z < 0) || (!isPlayer && direction.z > 0))
        {
            direction.z *= -1;
        }
    }

    if (collision.gameObject.tag == "Side")
    {
        direction.x *= -1;
    }
}
```

}

Enregistrez-le et retournez dans Unity. Avant de tester le code, pensez à cliquer sur la balle et à modifier la valeur de la variable speed pour indiquer 10 :



Cela permettra d'avoir une vitesse de déplacement non nulle. 10 étant une valeur qui me semble correcte dans notre cas.

Vous pouvez à présent tester votre jeu et vérifier que la balle fonctionne comme prévu. Si ce n'est pas le cas, reprenez ce chapitre avant de passer au suivant.

Code source

```
using System.Collections;

using System.Collections.Generic;

using UnityEngine;


public class PongBall : MonoBehaviour

{

  public float speed;

  public Vector3 direction;

  private float zMaxDistance = 15f;


  private void Start()

  {

    SetDirection();
```

```
}

void Update()
{
   transform.Translate(direction * speed * Time.deltaTime);

   // IA gange
   if (transform.position.z < -zMaxDistance && direction.z < 0)
   {
      SetDirection();
   }

   // Joueur gange
   if (transform.position.z > zMaxDistance && direction.z > 0)
   {
      SetDirection();
   }
}

public void SetDirection()
{
   transform.position = new Vector3(0, .5f, 0);
   direction = new Vector3(Random.Range(0.75f, 1.75f), 0, -1).normalized;
}

private void OnCollisionEnter(Collision collision)
{
   if (collision.gameObject.tag == "Bar")
```

```
    {
        bool isPlayer = collision.gameObject.GetComponent<PongBar>().isHumanPlayer;
        if ((isPlayer && direction.z < 0) || (!isPlayer && direction.z > 0))
        {
            direction.z *= -1;
        }
    }

    if (collision.gameObject.tag == "Side")
    {
        direction.x *= -1;
    }
  }
}
```

Script de déplacement de l'ordinateur

Actuellement, la raquette de l'ordinateur n'est contrôlée que via les flèches haut et bas de notre clavier. Ce n'est pas ce que nous souhaitons. Dans l'idée, c'est l'ordinateur qui doit contrôler sa raquette de façon autonome afin que l'on puisse l'affronter.

Je vous invite donc à créer un nouveau script C# que vous appellerez « PongAI ». Ce script sera l'intelligence artificielle de l'ordinateur. Une fois créé, ajoutez ce script à la raquette de l'ordinateur uniquement. Enfin, ouvrez le script pour pouvoir l'éditer.

Le code de notre IA sera relativement simple. Nous allons avoir besoin de 3 variables. La première permettra de stocker le transform de la balle. Cela nous permettra entre autres de connaître la position précise de la balle sur le terrain.

La seconde variable sera la latence. En quelques sortes c'est le temps de réaction de notre intelligence artificielle. C'est un paramètre qu'il sera possible d'ajuster afin de modifier la difficulté du jeu.

La dernière variable sera la distance à laquelle l'ordinateur sera capable de voir la balle. Lorsque la balle est dans le champ de vision de l'ordinateur, celui-ci peut réagir. Dans le cas contraire, il restera immobile.

Voilà donc les 3 variables avec leurs valeurs par défaut :

```
public Transform pongBall;

public float latency = 4f;

public float viewDistance = 20;
```

Ensuite dans la fonction update, nous allons utiliser la fonction Vector3.Distance afin de connaître la distance entre la raquette et la balle, ainsi nous serons en mesure de savoir si la balle est à portée de vue :

```
if (Vector3.Distance(transform.position, pongBall.position) < viewDistance)
{

}
```

Dans cette condition, nous allons modifier la position de la raquette via transform.position. Contrairement à ce que nous avons pu faire précédemment, nous utiliserons Vector3.MoveTowards afin que ce mouvement soit plus « humain » afin d'appliquer un effet de temps de latence, d'accélération et décélération du mouvement :

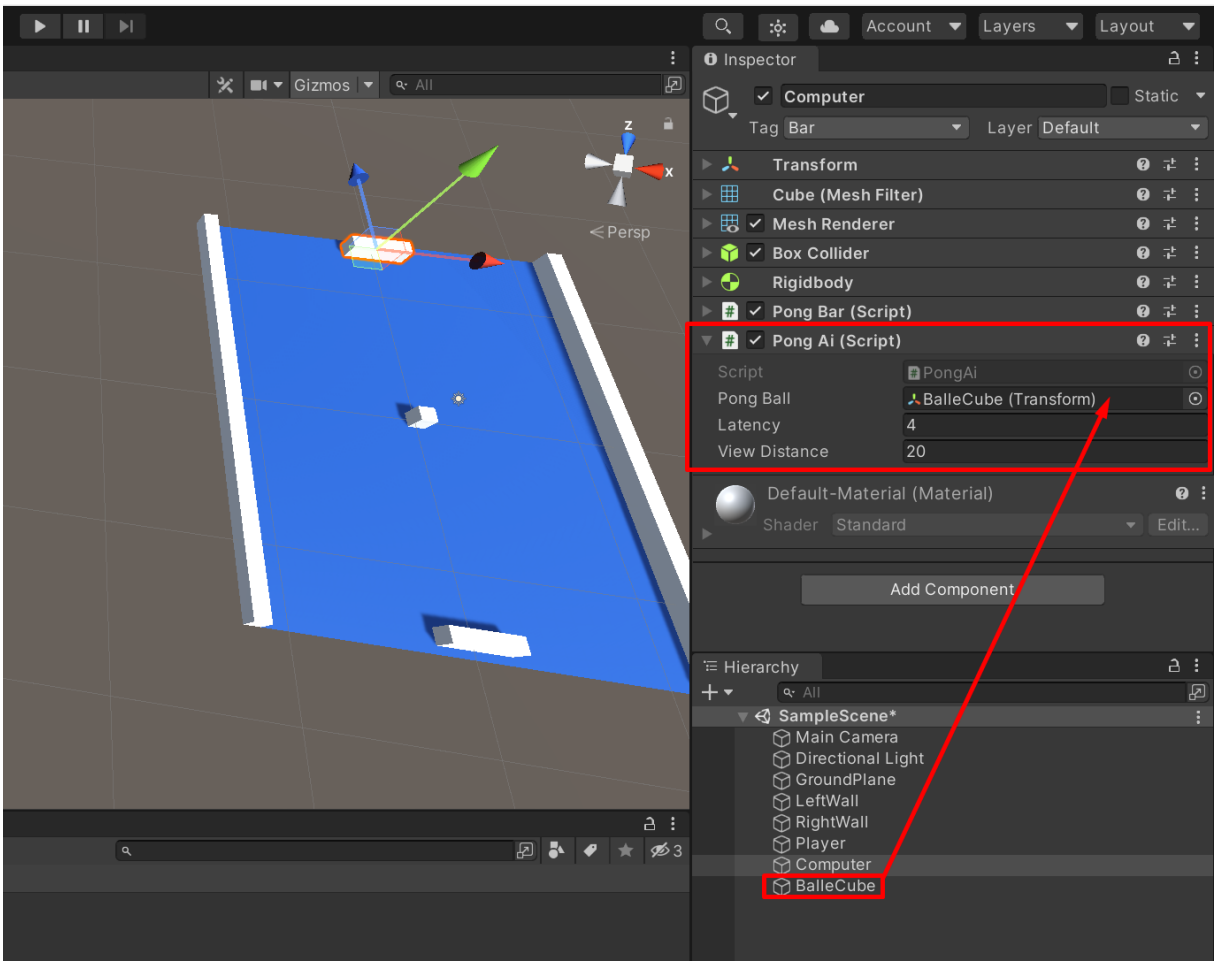
```
transform.position = Vector3.MoveTowards(

   transform.position,

   new Vector3(pongBall.transform.position.x, .5f, 14),

   latency * Time.deltaTime

);
```

Le code complet pour ce script est donc le suivant :

```
public class PongAi : MonoBehaviour
```

```
{
    public Transform pongBall;
    public float latency = 4f;
    public float viewDistance = 20;

    void Update()
    {
        if (Vector3.Distance(transform.position, pongBall.position) < viewDistance)
        {
            transform.position = Vector3.MoveTowards(
                transform.position,
                new Vector3(pongBall.transform.position.x, .5f, 14),
                latency * Time.deltaTime
            );
        }
    }
}
```

Vous pouvez sauvegarder le script, retourner dans Unity et vérifier que les valeurs dans l'inspector sont correctes. Vous devez notamment faire glisser la balle de la fenêtre hierarchy vers l'inspector afin de la lier au script via la variable « pongBall » que nous avons créée :



Une fois le tout configuré, vous pouvez tester votre projet. Pour cela cliquez sur Play et jouez contre l'ordinateur. Il devrait maintenant réagir de façon autonome.

Vous devez tester et ajuster les valeurs des variables via l'inspector. L'idée est de trouver des valeurs qui donnent un niveau de difficulté suffisant pour que le jeu soit amusant mais pas trop élevé pour que le joueur puisse gagner. Les valeurs dépendront de la vitesse de la balle. Au plus la balle est rapide, au plus l'IA devra réagir vite. Chacun sélectionnera les valeurs qui lui correspondent.

Il est possible de créer une IA plus poussée. Cela serait un bon exercice pour vous afin de vous améliorer. Je vais vous donner une idée que vous devriez être en mesure de mettre en place assez facilement :

- Créez une variable qui permettra de stocker le nombre de rebonds sur la raquette de l'ordinateur.
- Faites en sorte que votre ordinateur soit très fort (presque imbattable).
- Au bout d'un certain temps (par exemple tous les 10 rebonds), diminuez le temps de réaction de l'ordinateur.

- Votre objectif est de créer une sorte de jauge de fatigue. Au plus le temps passe, au moins l'ordinateur est réactif et au plus il est facile de le battre.
- Vous devrez donc modifier les valeurs des variables dynamiquement selon le nombre de rebonds.
- Vous pouvez créer une fonction publique dans le script de l'IA qui sera appelée par le script de la balle lors d'un rebond.

Vous avez tout en main pour réaliser cet exercice. Vous savez détecter des collisions, vous savez modifier la valeur d'une variable, vous savez créer des fonctions, utiliser les conditions. Bref toutes les notions ont été vues dans les sections précédentes. Je vous suggère de passer au prochain chapitre que lorsque vous aurez réussi cet exercice, cela prouvera que vous maîtrisez parfaitement ce qui a été vu jusqu'ici.

Si vous bloquez réellement sur cet exercice (après avoir bien essayé de votre côté), voici une solution possible.

Il faut modifier le code de « PongBall » et ajouter dans la fonction OnCollisionEnter, à l'intérieur de la condition qui tests si la balle touche la raquette cette nouvelle condition :

```
if(!isPlayer)
{
collision.gameObject.GetComponent<PongAi>().AddBounce();
}
```

Ce qui nous donne la fonction de collision complète suivante :

```
private void OnCollisionEnter(Collision collision)
  {
     if (collision.gameObject.tag == "Bar")
     {
        bool isPlayer = collision.gameObject.GetComponent<PongBar>().isHumanPlayer;
        if ((isPlayer && direction.z < 0) || (!isPlayer && direction.z > 0))
        {
           direction.z *= -1;
```

```
        }

        if(!isPlayer)
        {
            collision.gameObject.GetComponent<PongAi>().AddBounce();
        }
    }

    if (collision.gameObject.tag == "Side")
    {
        direction.x *= -1;
    }
}
```

Il faut ensuite modifier le script « PongIa » pour y ajouter la variable suivante :

```
public int nbShots = 0;
```

Ainsi que la fonction suivante :

```
public void AddBounce()
{
    nbShots++;
    if(nbShots >= 10)
    {
        nbShots = 0;
        latency -= 0.25f;
    }
}
```

Ce qui nous donne le code :

```
using UnityEngine;

public class PongAi : MonoBehaviour
{
   public Transform pongBall;
   public float latency = 4f;
   public float viewDistance = 20;
   public int nbShots = 0;

   void Update()
   {
      if (Vector3.Distance(transform.position, pongBall.position) < viewDistance)
      {
         transform.position = Vector3.MoveTowards(
            transform.position,
            new Vector3(pongBall.transform.position.x, .5f, 14),
            latency * Time.deltaTime
         );
      }
   }

   public void AddBounce()
   {
      nbShots++;
      if(nbShots >= 10)
      {
```

```
        nbShots = 0;
        latency -= 0.25f;
      }
    }
}
```

Pensez à sauvegarder, à modifier les variables dans l'inspector pour rendre l'IA plus forte et à tester afin de vérifier qu'elle s'épuise au cours du temps et qu'elle devient simple à battre au bout d'un certain temps. Pour tester plus rapidement, vous pouvez mettre 2 ou 3 rebonds au lieu des 10.

Code source

```
using UnityEngine;

public class PongAi : MonoBehaviour
{
  public Transform pongBall;
  public float latency = 4f;
  public float viewDistance = 20;
  public int nbShots = 0;

  void Update()
  {
    if (Vector3.Distance(transform.position, pongBall.position) < viewDistance)
    {
      transform.position = Vector3.MoveTowards(
        transform.position,
        new Vector3(pongBall.transform.position.x, .5f, 14),
        latency * Time.deltaTime
```

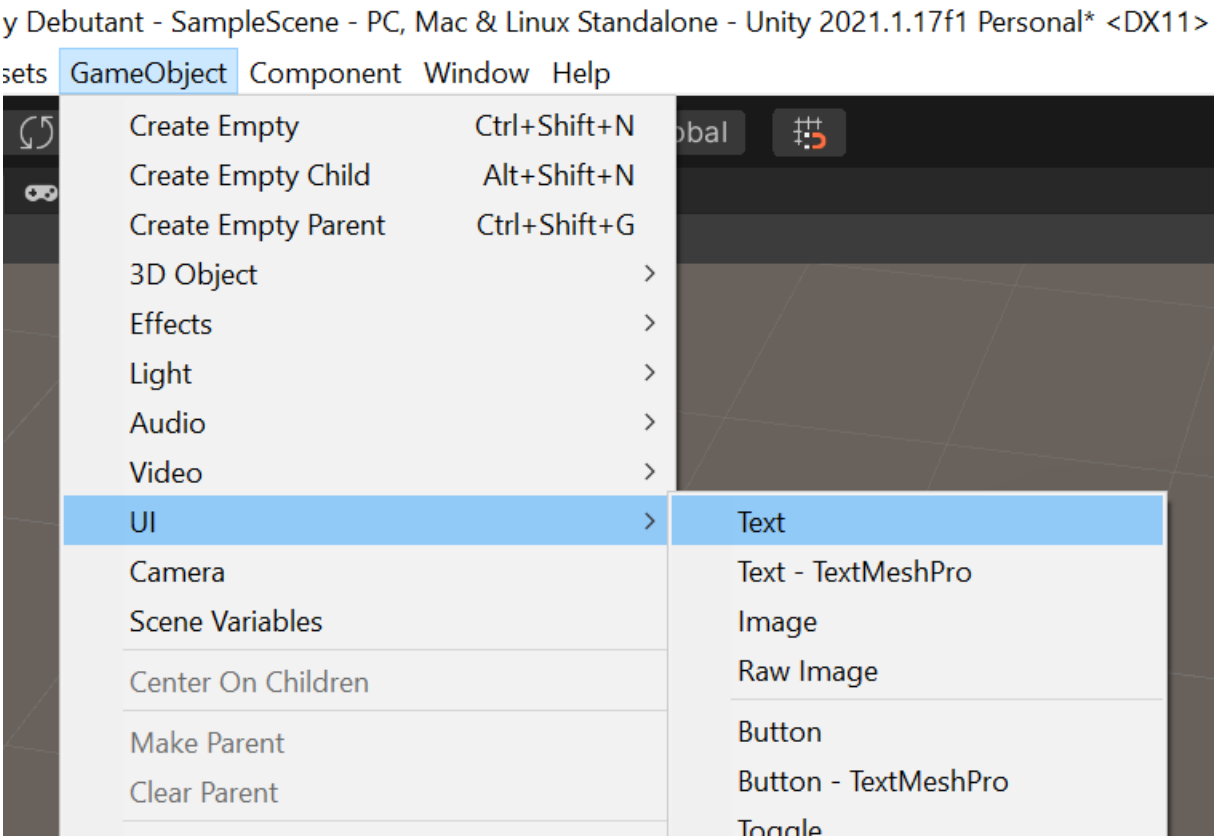
```
            );
        }
    }

    public void AddBounce()
    {
        nbShots++;
        if(nbShots >= 10)
        {
            nbShots = 0;
            latency -= 0.25f;
        }
    }
}
```

Calcul et affichage du score sur une IHM

Notre jeu est maintenant fonctionnel. Nous pouvons affronter l'ordinateur et jouer à l'infini à notre clone de Pong en 3D. Cependant, nous ne calculons pas le score. Très rapidement on oublie quel est le score et qui gagne. Nous allons donc voir comment afficher le score à l'écran afin qu'il soit plus simple d'identifier qui a le plus de points entre le joueur et l'ordinateur.

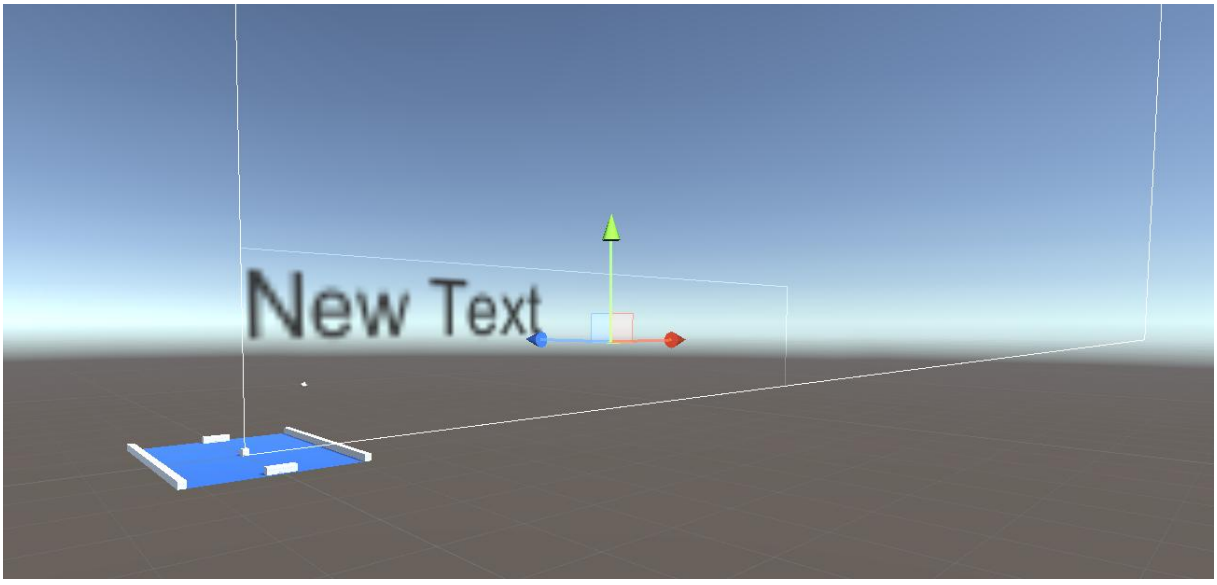
Pour afficher le score à l'écran, nous allons utiliser les outils d'UI (user interface) de Unity. Le principe est assez simple : Nous créons un canvas (une sorte de rectangle qui apparaît dans la scène) et dans ce canvas nous créons l'interface utilisateur. Tout ce qui se trouve à l'intérieur du canvas sera affiché à l'écran au-dessus du jeu 3D.

Nous allons faire cela afin de mieux visualiser le tout. Cliquez sur « GameObject / UI / text » :



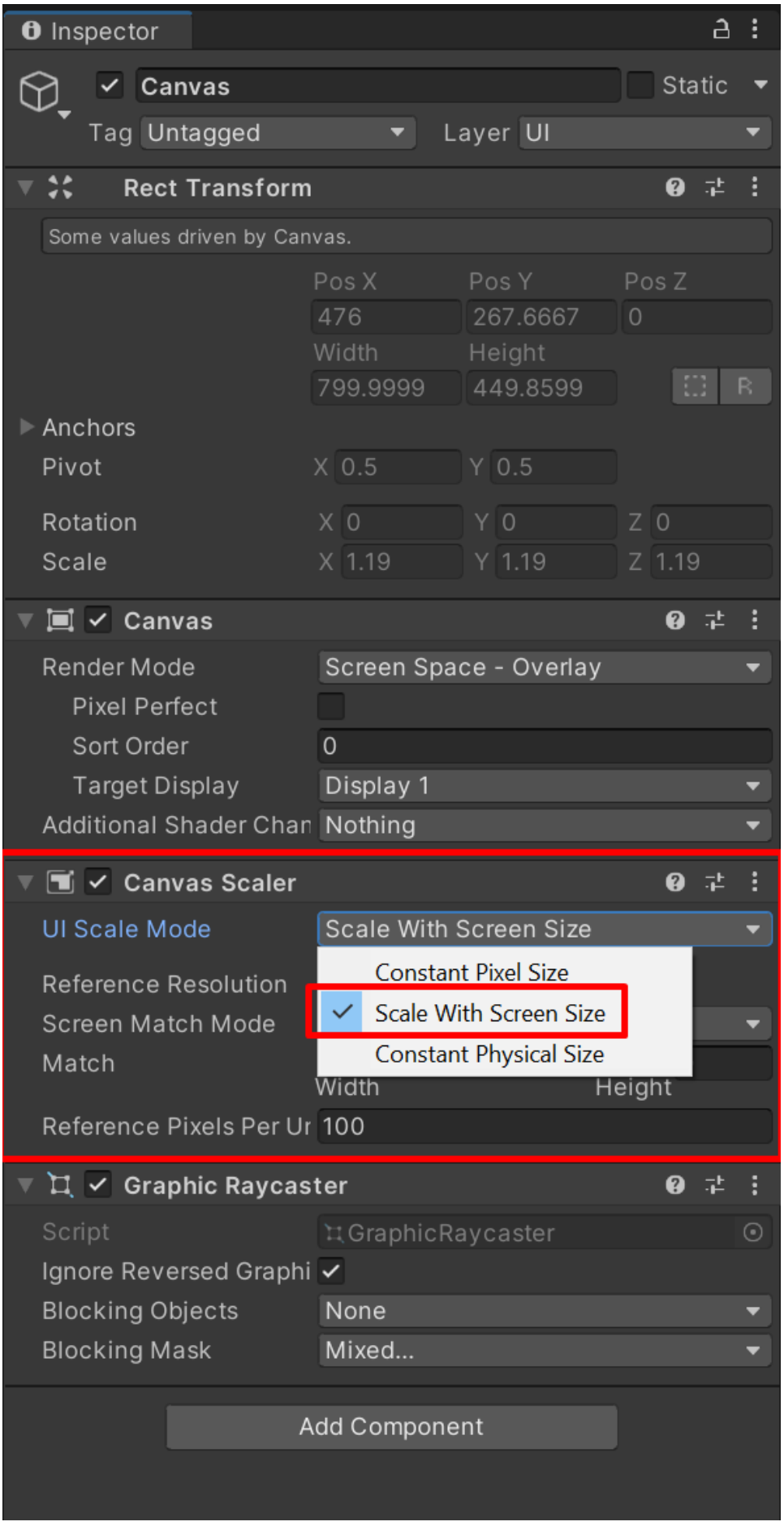
Cela aura pour effet de créer un texte mais aussi un Canvas automatiquement pour nous (vous pouvez constater les changements dans la hierarchy).

Si vous regardez la scène, vous remarquerez un rectangle blanc géant :



C’est notre canvas. Le texte apparaît en bas à gauche du canvas. Si le texte est dans le rectangle blanc alors il apparaîtra dans le jeu.

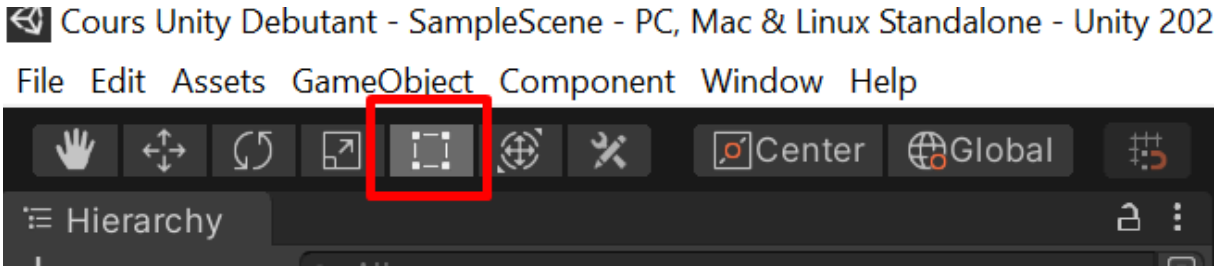
Cliquez sur le Canvas (via la hierarchy) et ajustez le réglage « Scale Mode » depuis l’inspector afin de choisir « Scale with screen size » :


Inspector
Canvas
Static
Tag Untagged
Layer UI
Rect Transform
Some values driven by Canvas.
Pos X
Pos Y
Pos Z
476
267.6667
0
Width
Height
799.9999
449.8599
Anchors
Pivot
X 0.5
Y 0.5
Rotation
X 0
Y 0
Z 0
Scale
X 1.19
Y 1.19
Z 1.19
Canvas
Render Mode
Screen Space - Overlay
Pixel Perfect
Sort Order
0
Target Display
Display 1
Additional Shader Chan
Nothing
Canvas Scaler
UI Scale Mode
Scale With Screen Size
Reference Resolution
Screen Match Mode
Match
Width
Height
Reference Pixels Per Ur
100
Constant Pixel Size
Scale With Screen Size
Constant Physical Size
Graphic Raycaster
Script
GraphicRaycaster
Ignore Reversed Graphi
Blocking Objects
None
Blocking Mask
Mixed...
Add Component

Ce changement permettra de faire en sorte que les proportions du canvas soient en pourcentage de la taille de l'écran. En d'autres mots, même si vous jouez sur petit écran vous verrez les éléments à l'écran avec une taille proportionnelle.

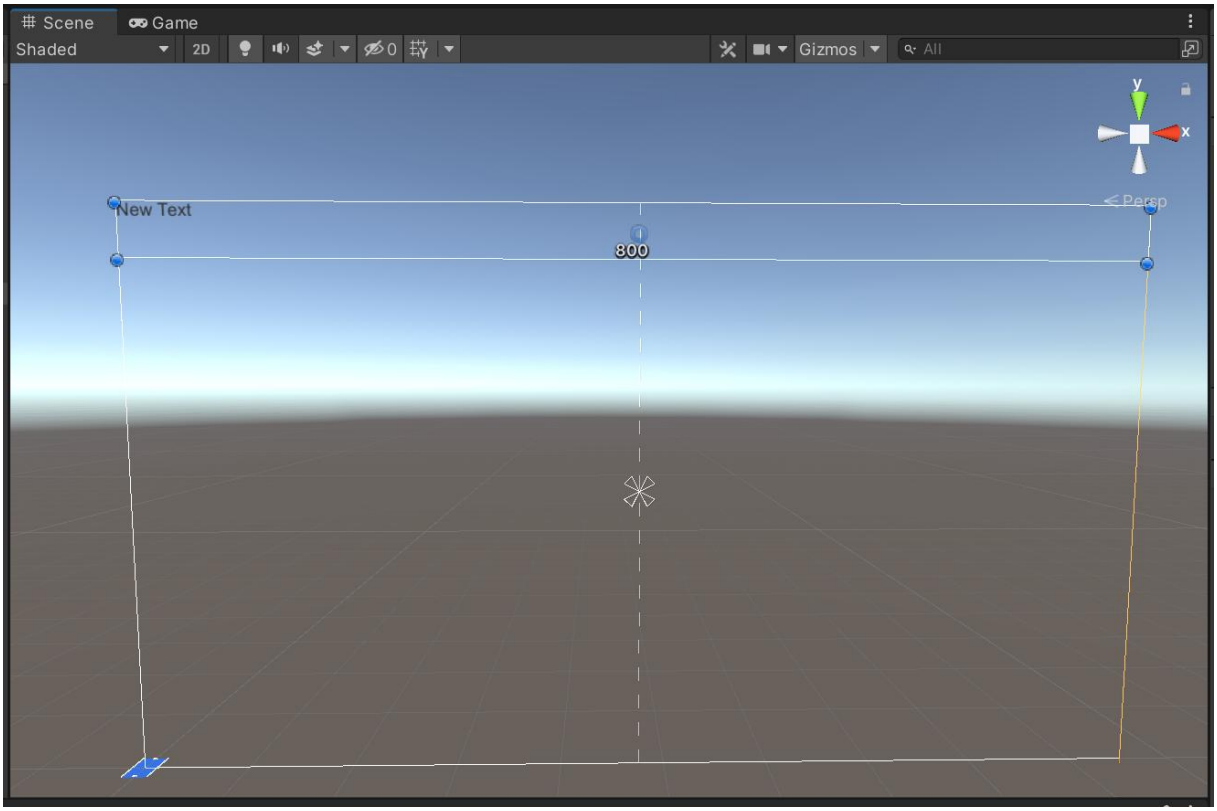
Je sais que tout cela fait beaucoup d'informations. J'essaye de donner des explications simples et concises mais c'est normal de ne pas forcément tout retenir. L'idée c'est de savoir que ça existe, que c'est possible et ensuite vous pourrez plus facilement rechercher des informations sur le net en tapant les bons mots-clés pour retrouver une information le moment venu.

Utilisez l'icône suivant de la barre d'outils :



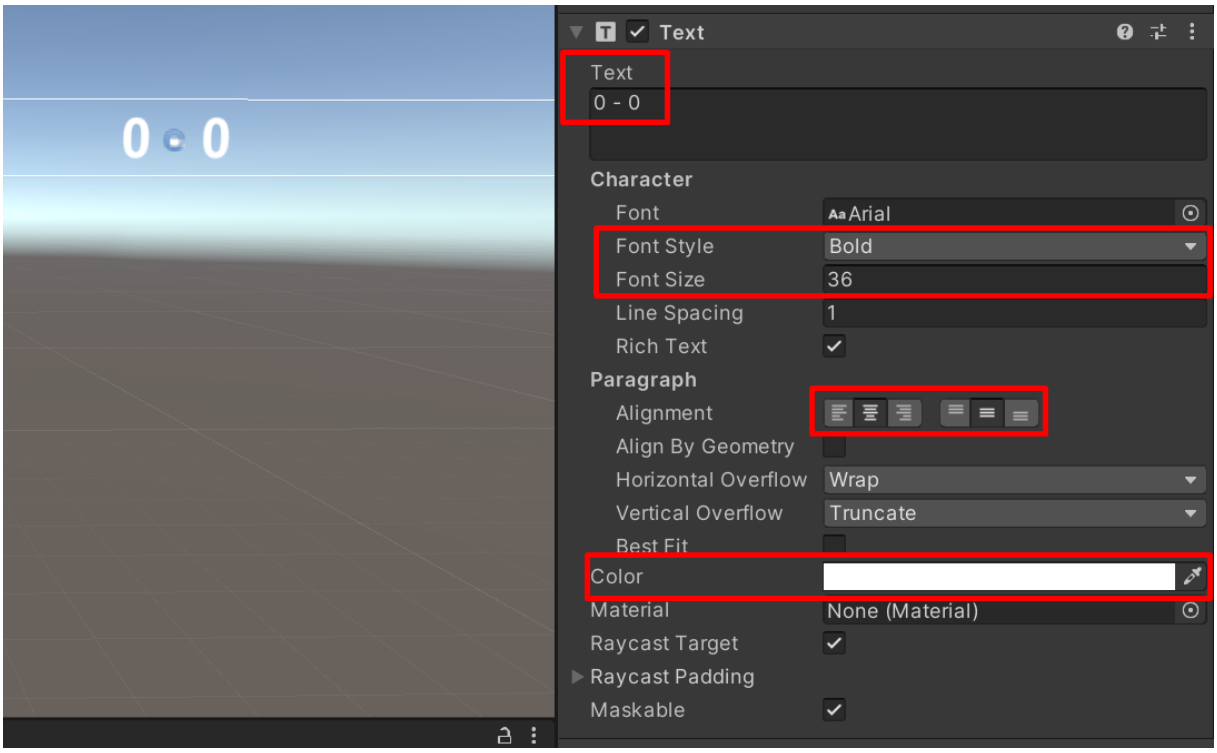
Cet outil permet de déplacer, tourner et changer la taille d'un élément UI. C'est donc spécifique aux éléments qui se trouvent dans un canvas.

Cliquez ensuite sur le texte et déplacez-le en haut du canvas. Augmentez la taille du rectangle du texte afin de l'étendre sur toute la largeur du canvas :



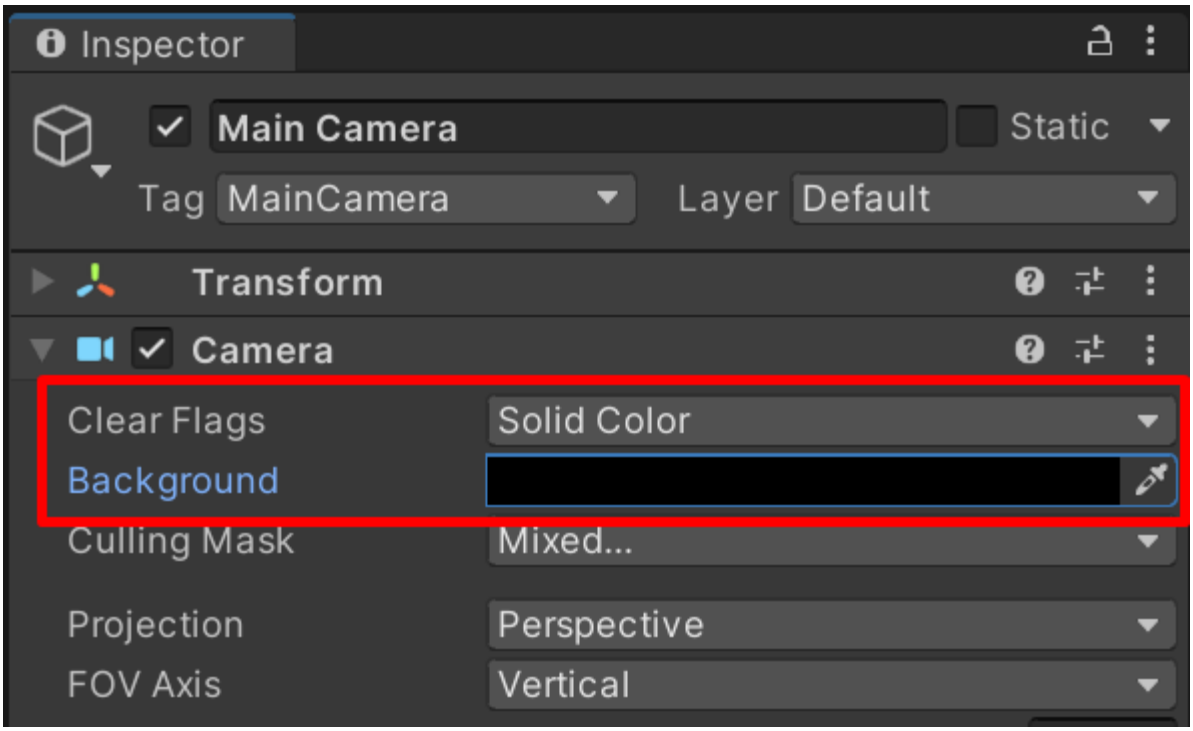
Vous pouvez utiliser les petits ronds bleus (visibles sur ma capture) pour agrandir le rectangle et le positionner comme moi.

Toujours avec le texte sélectionné, modifiez ses propriétés via l'inspector. Vous avez accès à un certain nombre de réglages comme la valeur du texte, son style, sa taille, sa couleur, son indentation etc. Modifiez le tout avec les mêmes réglages que moi pour aboutir au même résultat :



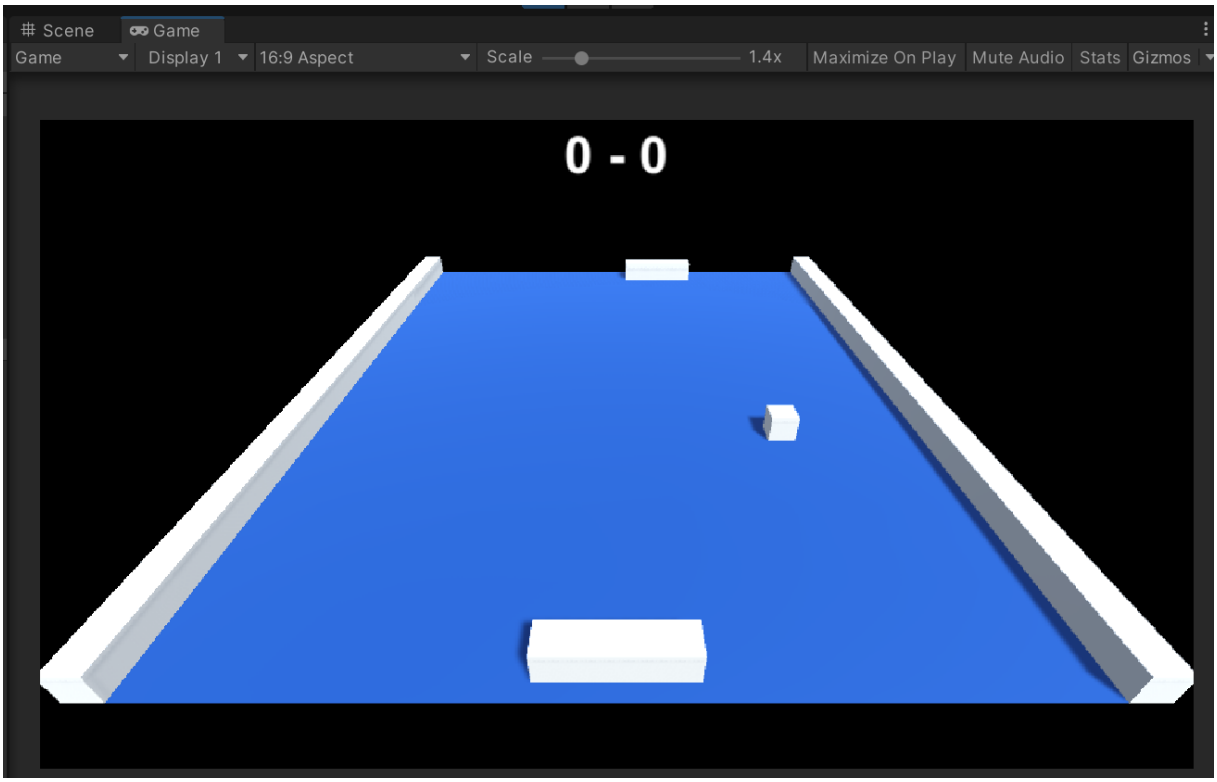
Pour vous faciliter la tâche, j'ai mis en évidence les différentes propriétés à modifier.

Nous allons également cliquer sur la caméra et modifier ces réglages afin d'afficher un fond de couleur unie (fond noir) :



Cela nous permettra de rendre le texte beaucoup plus lisible à l'écran.

Sauvegardez et testez le jeu. Vous devriez voir le texte apparaître à l'écran :



Evidement ce texte n'est pas dynamique, le score reste à 0 – 0 mais nous progressons. Nous avons maintenant un texte qui s'affiche et il ne nous reste plus qu'à le modifier par script.

Retournez dans le script « PongBall » et ajoutez deux nouvelles variables :

```
private int scorePlayer = 0;

private int scoreComputer = 0;
```

Ces variables permettront de stocker le score des deux joueurs.

Au moment ou l'IA marque un but (cas géré dans la condition de la fonction Update), augmentez son score de cette façon :

```
scoreComputer++;
```

Si c'est le joueur qui marque alors utilisez cette ligne de code :

```
scorePlayer++;
```

Maintenant que le score est bien calculé, nous devons l'afficher. Créez une variable qui permettra de stocker le texte :

```
public Text scoreText;
```

Le type Text est spécifique aux composants UI de Unity. Pour pouvoir utiliser (par script) les composants UI, nous devons ajouter une instruction using tout en haut du script :

```
using UnityEngine.UI;
```

Modifiez ensuite la fonction SetDirection qui réinitialise le jeu à chaque but. Ajoutez la ligne de code suivante qui permet de modifier le texte pour afficher les deux scores :

```
scoreText.text = scorePlayer.ToString() + " - " + scoreComputer.ToString();
```

Ainsi, à chaque but le score sera mis à jour sur l'interface utilisateur. Le nouveau code complet du script PongBall est le suivant :

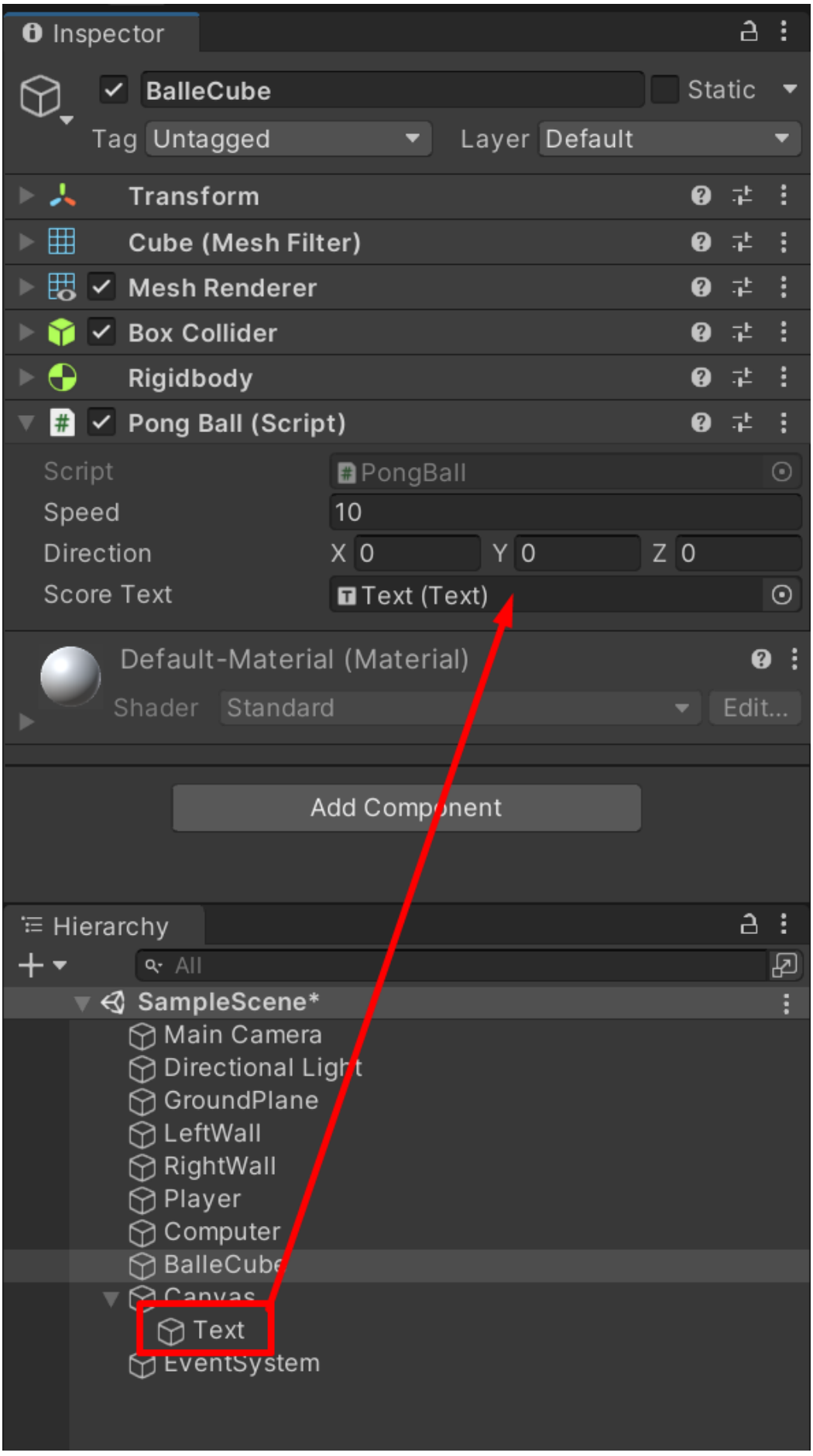
```
using System.Collections;

using System.Collections.Generic;

using UnityEngine;
```

```
using UnityEngine.UI;

public class PongBall : MonoBehaviour
{
    public float speed;
    public Vector3 direction;
    public Text scoreText;
    private float zMaxDistance = 15f;
    private int scorePlayer = 0;
    private int scoreComputer = 0;

    private void Start()
    {
        SetDirection();
    }

    void Update()
    {
        transform.Translate(direction * speed * Time.deltaTime);

        // IA gange
        if (transform.position.z < -zMaxDistance && direction.z < 0)
        {
            scoreComputer++;
            SetDirection();
        }

        // Joueur gange
```

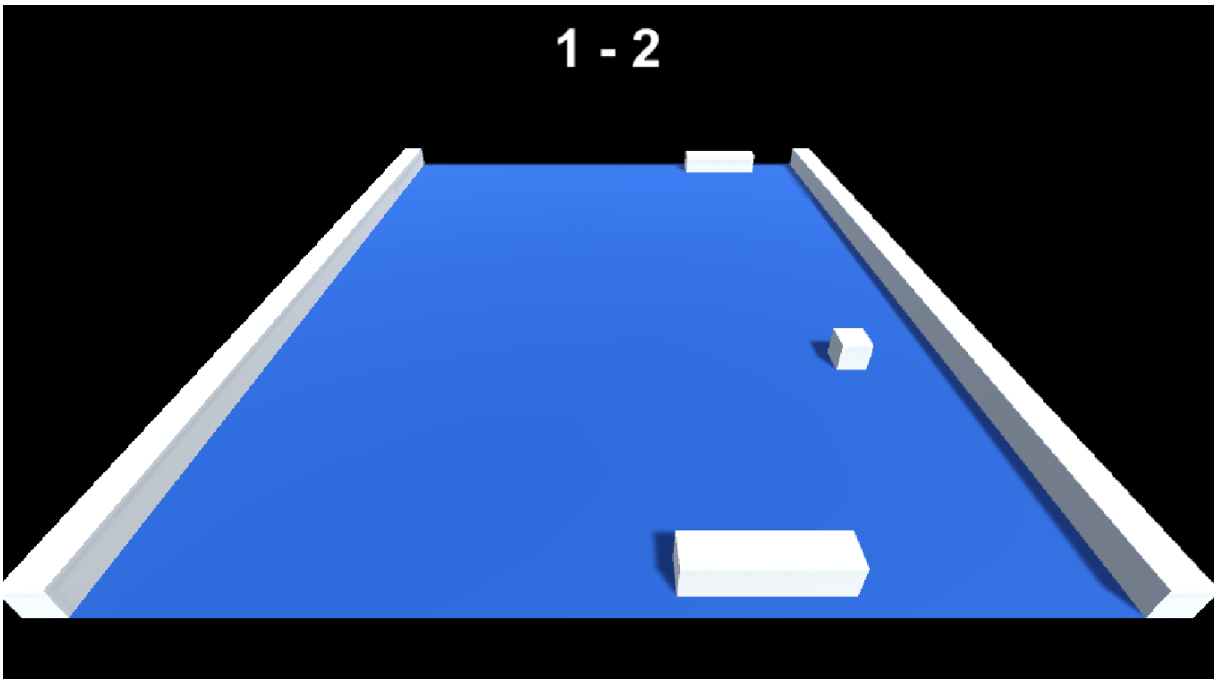
```
    if (transform.position.z > zMaxDistance && direction.z > 0)
    {
        scorePlayer++;
        SetDirection();
    }
}

public void SetDirection()
{
    scoreText.text = scorePlayer.ToString() + " - " + scoreComputer.ToString();
    transform.position = new Vector3(0, .5f, 0);
    direction = new Vector3(Random.Range(0.75f, 1.75f), 0, -1).normalized;
}

private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "Bar")
    {
        bool isPlayer = collision.gameObject.GetComponent<PongBar>().isHumanPlayer;
        if ((isPlayer && direction.z < 0) || (!isPlayer && direction.z > 0))
        {
            direction.z *= -1;
        }

        if(!isPlayer)
        {
            collision.gameObject.GetComponent<PongAi>().AddBounce();
        }
```

```
        }


        if (collision.gameObject.tag == "Side")
        {
            direction.x *= -1;
        }
    }
}
```

Sauvegardez le tout et retournez dans Unity. Vous devez à présent cliquer sur la balle et ajuster ses variables dans l'inspector afin de glisser le texte dans la variable associée :


Inspector
BalleCube
Static
Tag Untagged
Layer Default
Transform
Cube (Mesh Filter)
Mesh Renderer
Box Collider
Rigidbody
Pong Ball (Script)
Script PongBall
Speed 10
Direction X 0 Y 0 Z 0
Score Text Text (Text)
Default-Material (Material)
Shader Standard
Edit...
Add Component
Hierarchy
All
SampleScene*
Main Camera
Directional Light
GroundPlane
LeftWall
RightWall
Player
Computer
BalleCube
Canvas
Text
EventSystem

Le script aura alors accès au texte et pourra le modifier comme décrit dans notre code. Vous pouvez à présent tester le jeu et vérifier que le score est correctement mis à jour en temps réel :



Si c'est le cas, bravo ! Votre jeu est prêt. Vous pouvez bien sûr améliorer celui-ci, ajouter des fonctionnalités et l'élaborer un peu plus mais la base est là. Dans le prochain chapitre nous allons le peaufiner.

Code source

```
using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;


public class PongBall : MonoBehaviour

{

   public float speed;

   public Vector3 direction;

   public Text scoreText;

   private float zMaxDistance = 15f;

   private int scorePlayer = 0;
```

```
private int scoreComputer = 0;

private void Start()
{
   SetDirection();
}

void Update()
{
   transform.Translate(direction * speed * Time.deltaTime);

   // IA gange
   if (transform.position.z < -zMaxDistance && direction.z < 0)
   {
      scoreComputer++;
      SetDirection();
   }

   // Joueur gange
   if (transform.position.z > zMaxDistance && direction.z > 0)
   {
      scorePlayer++;
      SetDirection();
   }
}

public void SetDirection()
{
```

```
        scoreText.text = scorePlayer.ToString() + " - " + scoreComputer.ToString();
        transform.position = new Vector3(0, .5f, 0);
        direction = new Vector3(Random.Range(0.75f, 1.75f), 0, -1).normalized;
    }

    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.tag == "Bar")
        {
            bool isPlayer = collision.gameObject.GetComponent<PongBar>().isHumanPlayer;
            if ((isPlayer && direction.z < 0) || (!isPlayer && direction.z > 0))
            {
                direction.z *= -1;
            }

            if(!isPlayer)
            {
                collision.gameObject.GetComponent<PongAi>().AddBounce();
            }
        }

        if (collision.gameObject.tag == "Side")
        {
            direction.x *= -1;
        }
    }
}
```

Amélioration du jeu (Effets visuels)

Dans cette section, nous allons rajouter quelques effets visuels au projet afin de l'améliorer et de le rendre plus plaisant à regarder. Je vais partager avec vous quelques astuces réutilisables dans tous vos futurs projets.

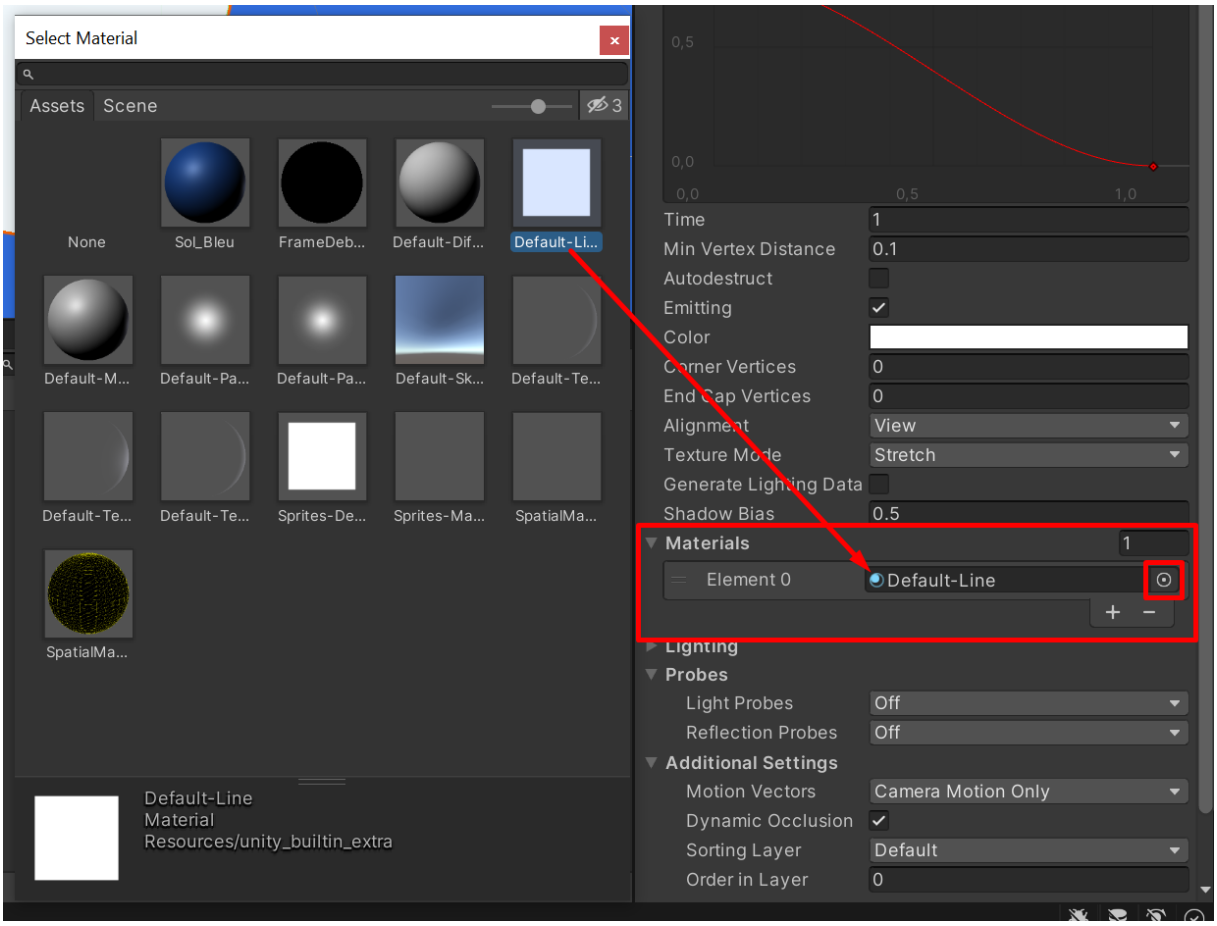
Commençons par ajouter une trainée lumineuse derrière la balle. Pour cela, cliquez sur celle-ci et utilisez le bouton « Add Component » de l'inspector afin de rechercher et de sélectionner « Trail Renderer ». Ce composant permet de faire apparaître une lueur derrière un objet en mouvement. Nous allons devoir configurer ce Trail Renderer afin d'ajuster les paramètres.

Commencez par modifier la ligne rouge qui apparaît. Pour la valeur width, mettez 0.8. Ensuite, double-cliquez sur cette courbe afin d'y ajouter un nouveau point. Ce point, placez-le en bas à droite afin de faire diminuer la valeur au cours du temps :

Ce réglage permettra d'avoir une trainée lumineuse dont la taille diminuera au cours du temps afin qu'elle s'estompe progressivement.

Modifiez aussi la valeur « Time » juste en dessous afin de passer de 5 à 1 pour la durée d'affichage de l'effet.

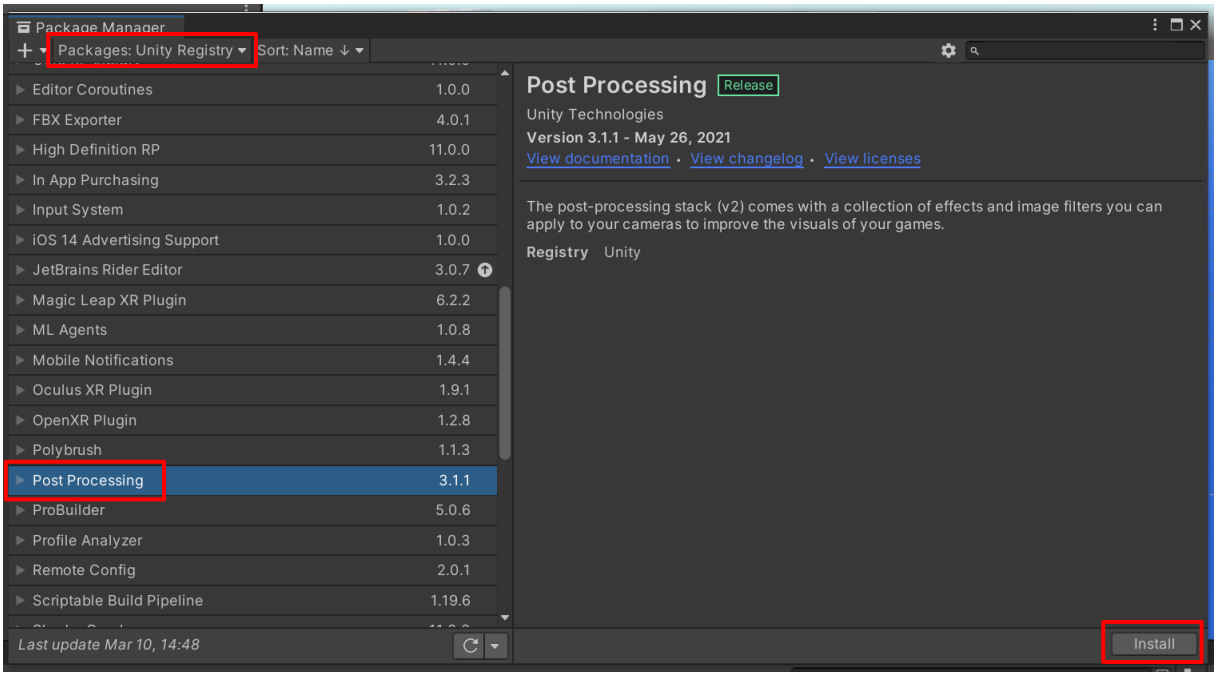
Enfin, modifiez le material en cliquant sur le petit rond de sélection (voir la capture suivante) et glissez le material « Default Line » dans la variable :



Vous pouvez maintenant tester votre projet et constater que l'effet visuel apparaît bien :

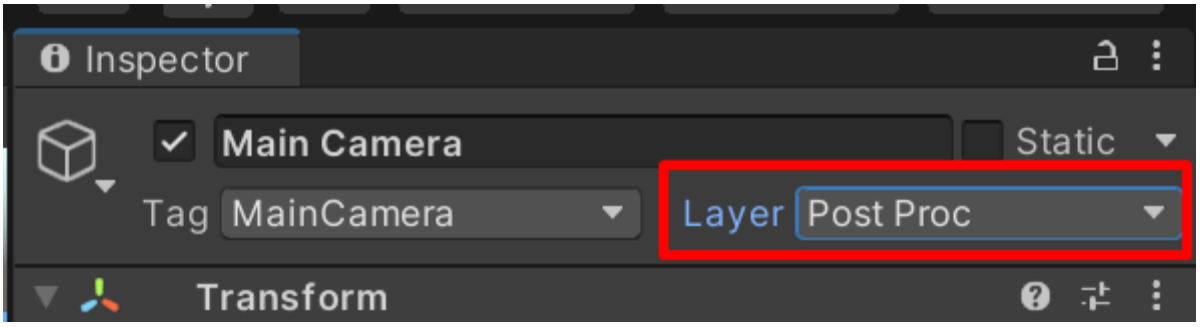
Nous allons maintenant appliquer des effets de post-traitement. Pour cela, cliquez sur le menu « Window / Package Manager ». Une fenêtre s'ouvre alors. Sélectionnez « Unity

Registry » dans la liste déroulante que je mets en évidence sur la capture suivante puis sélectionnez et installez le package « Post Processing » :



Cela va permettre d'ajouter les fonctionnalités de post-processing. Ces fonctionnalités permettent d'appliquer des effets visuels sur le rendu final du jeu.

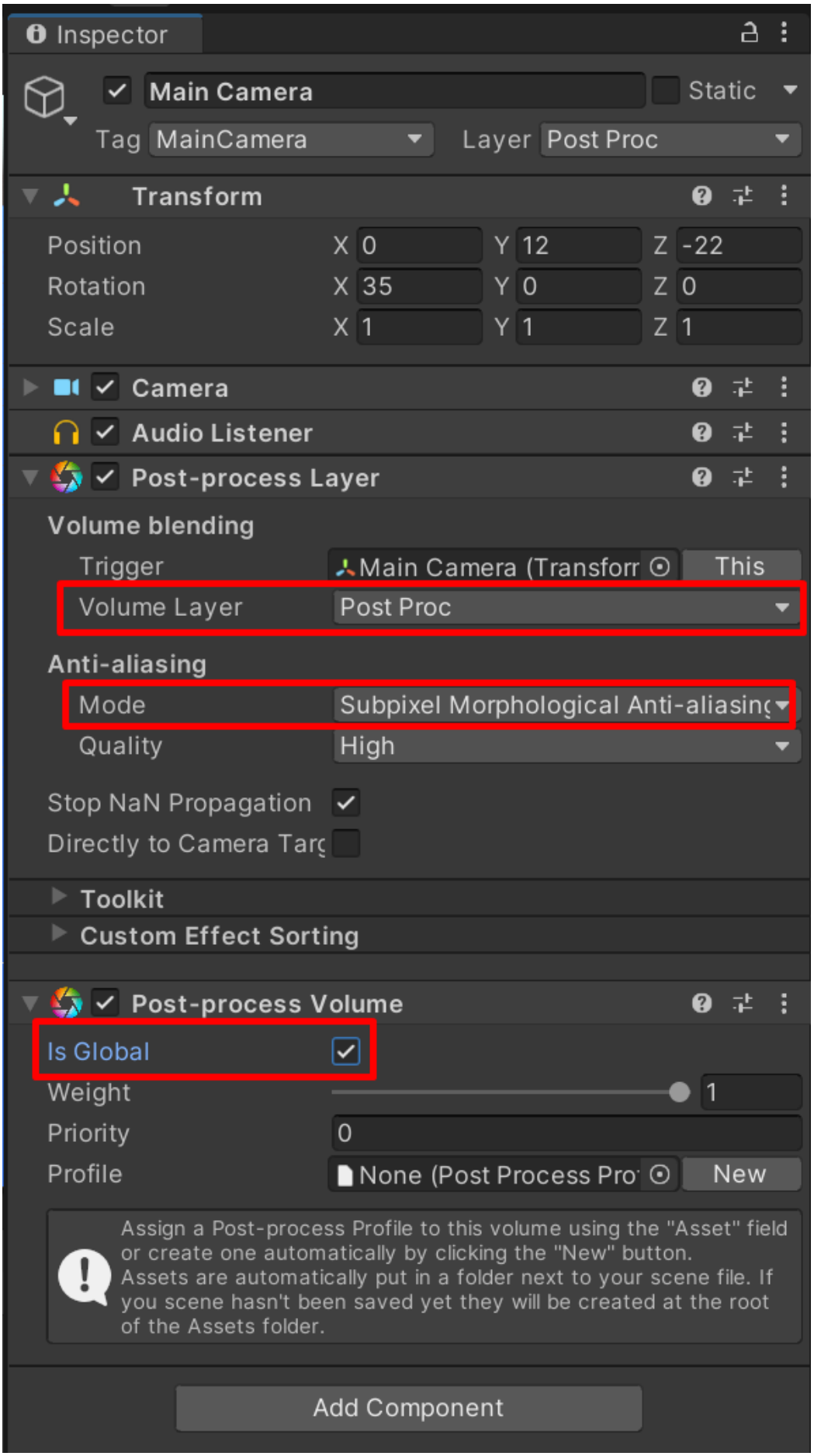
Cliquez sur la caméra et placez-la dans le layer « Post Proc » que vous aurez créé au préalable. La création d'un layer se déroule comme la création d'un Tag :



Cliquez ensuite sur le bouton « Add Component » pour ajouter un « Post-process Layer » et un « Post-process Volume » à la caméra.

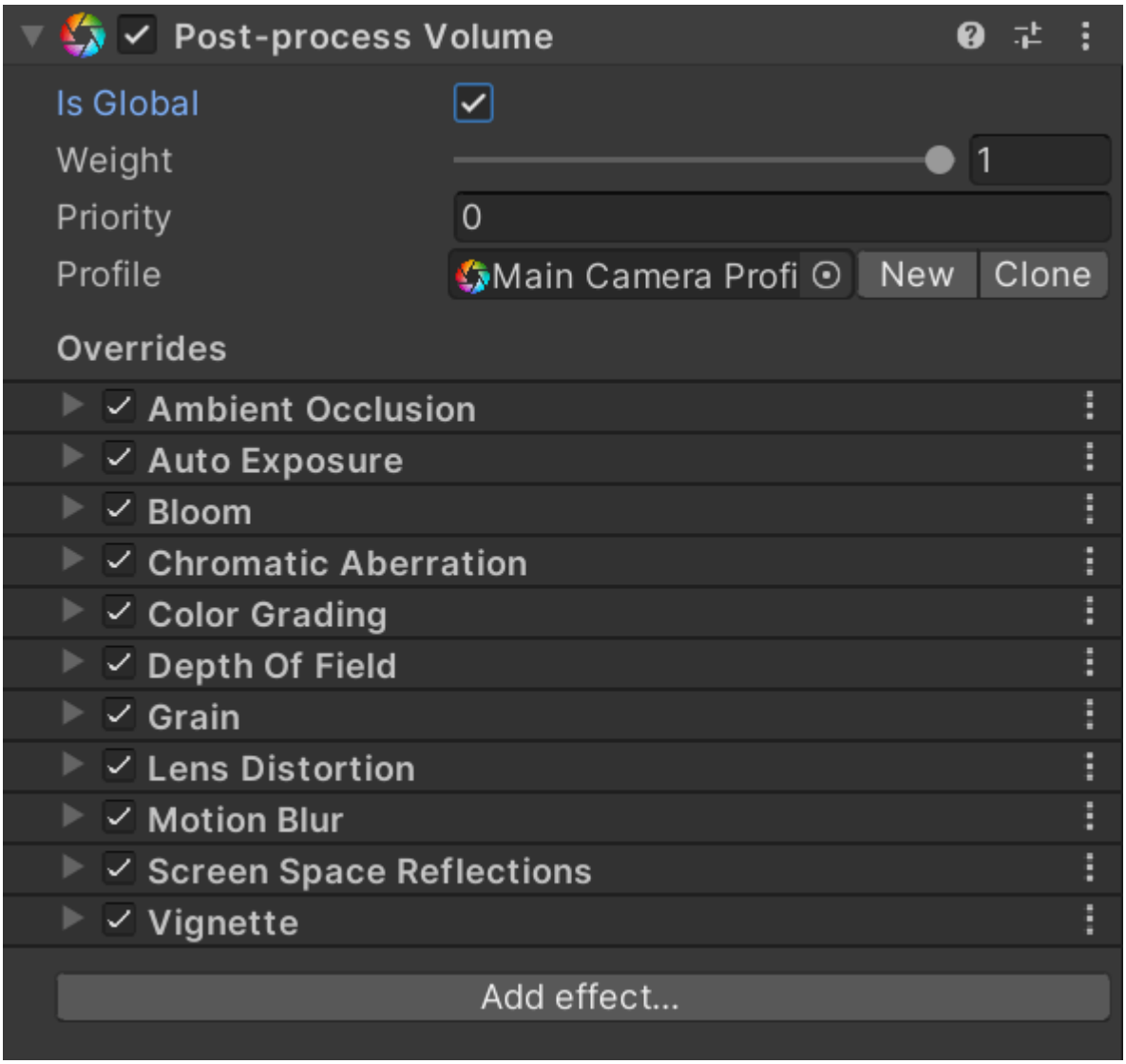
Pour la propriété « Volume Layer » sélectionnez « Post Proc ». Pour l'anti-aliasing, sélectionnez SMAA.

Pour le volume, cochez « Is Global ». Voilà les réglages que vous devez avoir à ce stade :


Inspector
Main Camera
Static
Tag MainCamera
Layer Post Proc
Transform
Position X 0 Y 12 Z -22
Rotation X 35 Y 0 Z 0
Scale X 1 Y 1 Z 1
Camera
Audio Listener
Post-process Layer
Volume blending
Trigger Main Camera (Transforr This
Volume Layer Post Proc
Anti-aliasing
Mode Subpixel Morphological Anti-aliasing
Quality High
Stop NaN Propagation
Directly to Camera Tarç
Toolkit
Custom Effect Sorting
Post-process Volume
Is Global
Weight 1
Priority 0
Profile None (Post Process Pro New
Assign a Post-process Profile to this volume using the "Asset" field or create one automatically by clicking the "New" button.
Assets are automatically put in a folder next to your scene file. If you scene hasn't been saved yet they will be created at the root of the Assets folder.
Add Component

Tout ceci va permettre d'appliquer les effets à la caméra car celle-ci est dans le layer sur lequel les effets vont s'appliquer. L'anti-aliasing permet de lisser les arêtes des modèles 3D. Is Global permet d'appliquer les effets sur la globalité de la scène.

Cliquez ensuite sur le bouton « New » à côté de « Profile » dans le Post-process Volume afin de créer un volume. Ce volume permet de lister et de configurer les effets que vous souhaitez appliquer. Pour ajouter des effets, après avoir créé ce volume, cliquez sur le bouton « Add effect » pour aller chercher un effet dans la liste. Je vous propose de tous les sélectionner afin que l'on fasse un tour complet. Vous devriez donc avoir ce résultat :

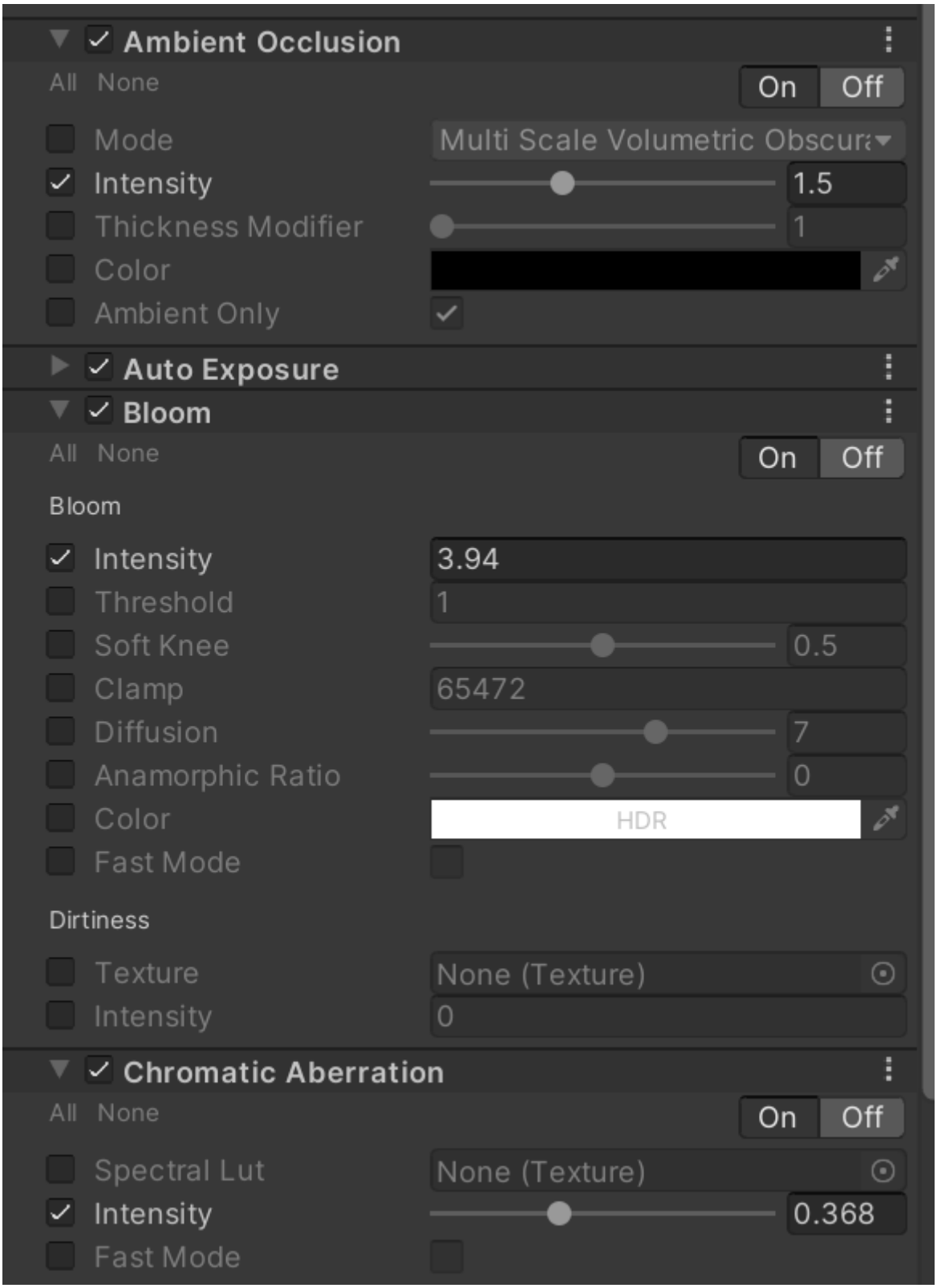


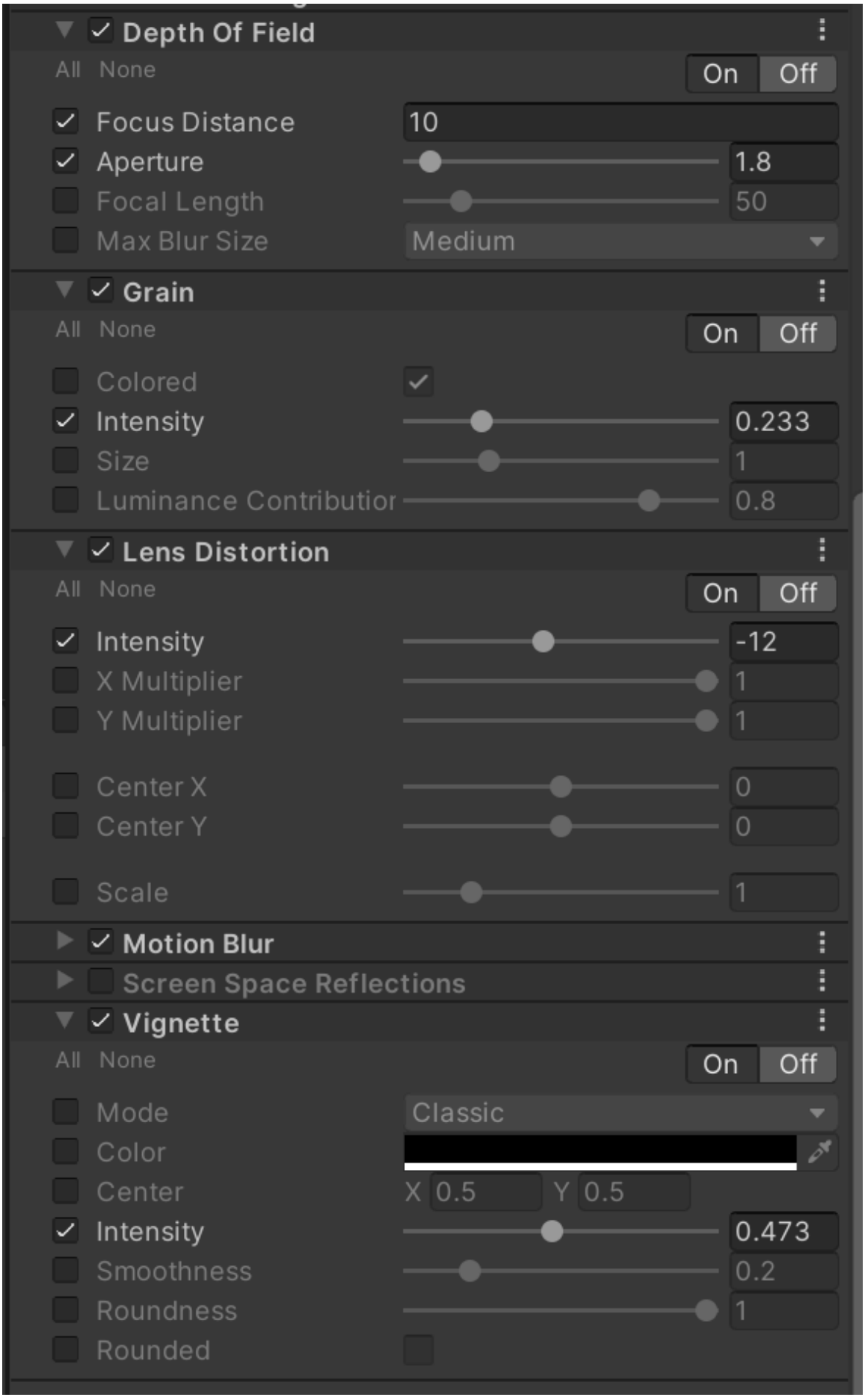
Par défaut, même s'ils sont ajoutés, ces effets n'ont aucun impact. Vous devez un à un les déplier et cocher chaque sous-paramètre individuellement et modifier les valeurs comme bon vous semble. Il existe beaucoup d'effets et de paramètres, je ne pourrai pas tout détailler ici mais vous avez de la documentation en ligne pour en savoir plus : https://docs.unity3d.com/Manual/PostProcessingOverview.html

Je vais brièvement décrire les effets :

- Ambient occlusion ajoute des sortes d'ombres autour des arêtes.
- Auto exposure simule l'éblouissement de nos yeux quand on sort d'une pièce sombre pour aller dans un endroit éclairé.
- Bloom crée une lueur (glowing). Utile par exemple pour la flamme d'une torche, la vitre d'un pare-brise, un néon etc.
- Chromatic aberration crée un effet de séparation de la lumière blanche en 3 couleurs primaire (comme sur les vieilles caméras).
- Color Grading permet de modifier les couleurs, je détaillerai cela plus tard.
- Depth of field crée un effet de profondeur permettant de flouter l'arrière-plan (ou l'avant-plan).
- Grain ajoute du grain à l'image comme sur les vieilles vidéos cassettes.
- Lens distortion permet de « tordre » l'image comme par exemple l'effet fish eye.
- Motion blur ajoute un flou de mouvement.
- Screen space reflection permet de mettre en place des réflections par exemple dans une flaque d'eau au sol.
- Vignette crée un effet de vignetage et assombrit les contours de l'image.

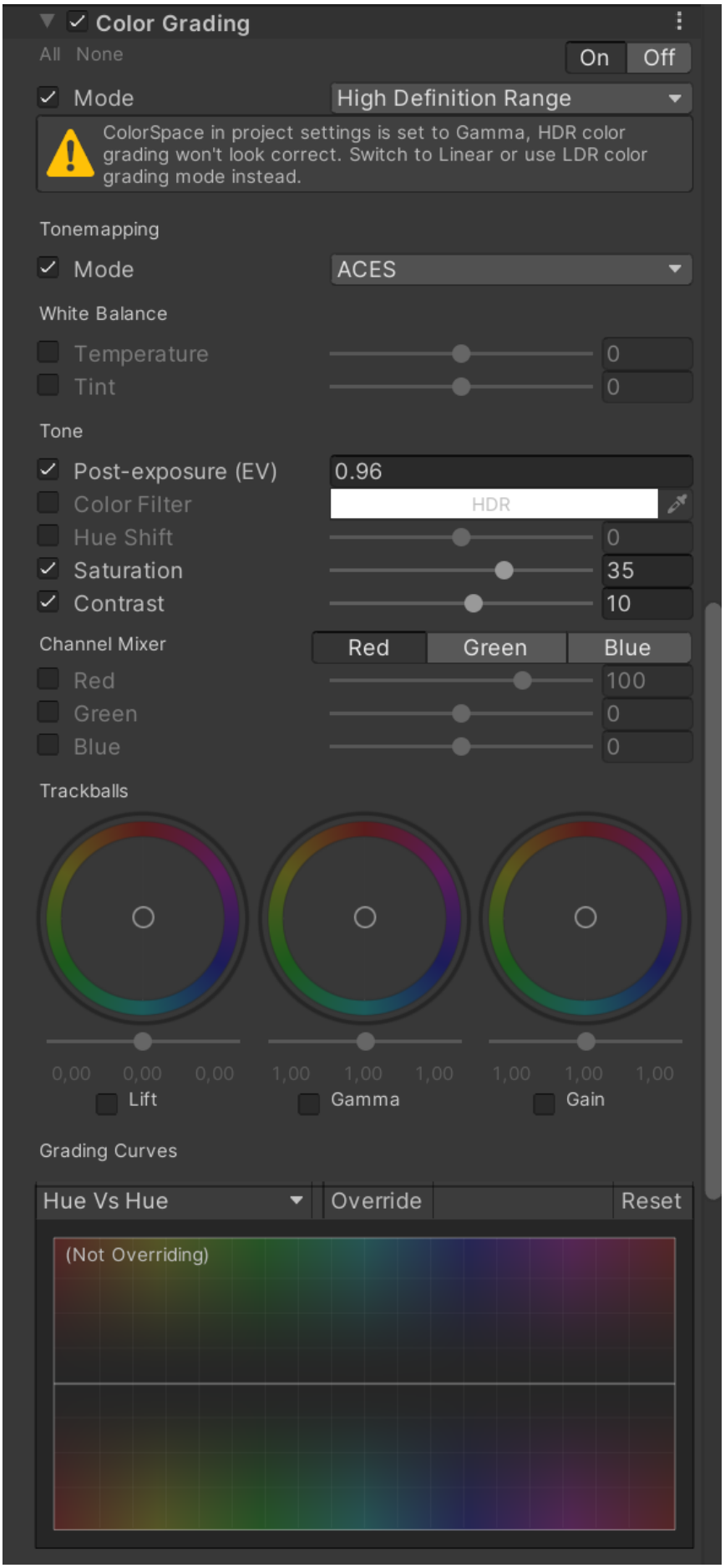
Voici un exemple de paramétrage (hors color grading) :


Ambient Occlusion
All None
On Off
Mode
Multi Scale Volumetric Obscura
Intensity
1.5
Thickness Modifier
1
Color
Ambient Only
Auto Exposure
Bloom
All None
On Off
Bloom
Intensity
3.94
Threshold
1
Soft Knee
0.5
Clamp
65472
Diffusion
7
Anamorphic Ratio
0
Color
HDR
Fast Mode
Dirtiness
Texture
None (Texture)
Intensity
0
Chromatic Aberration
All None
On Off
Spectral Lut
None (Texture)
Intensity
0.368
Fast Mode

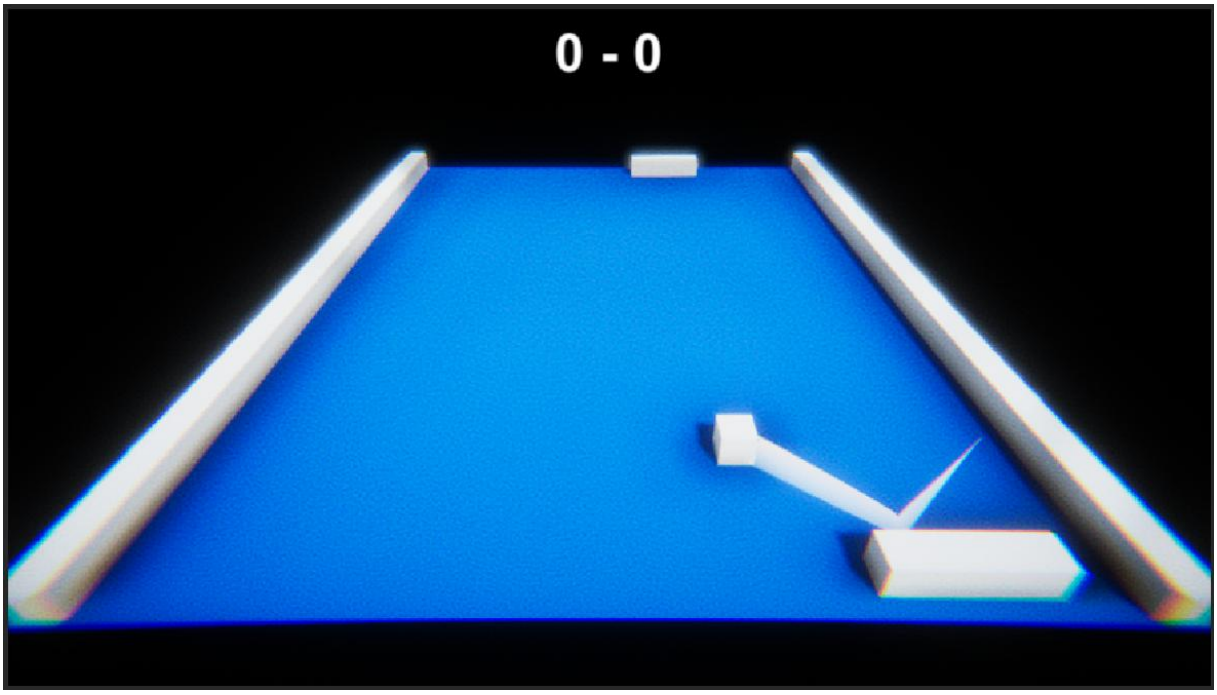

Depth Of Field
All None
On Off
Focus Distance
10
Aperture
1.8
Focal Length
50
Max Blur Size
Medium
Grain
All None
On Off
Colored
Intensity
0.233
Size
1
Luminance Contributior
0.8
Lens Distortion
All None
On Off
Intensity
-12
X Multiplier
1
Y Multiplier
1
Center X
0
Center Y
0
Scale
1
Motion Blur
Screen Space Reflections
Vignette
All None
On Off
Mode
Classic
Color
Center
X 0.5
Y 0.5
Intensity
0.473
Smoothness
0.2
Roundness
1
Rounded

Vous pouvez jouer avec ces réglages. Il n’est pas nécessaire de copier exactement les mieux, vous pouvez vous en inspirer mais paramétrez votre projet comme bon vous semble. Vous pouvez prévisualiser le rendu en direct dans Unity. L’idée est de modifier le rendu du jeu selon ce qui vous plait.

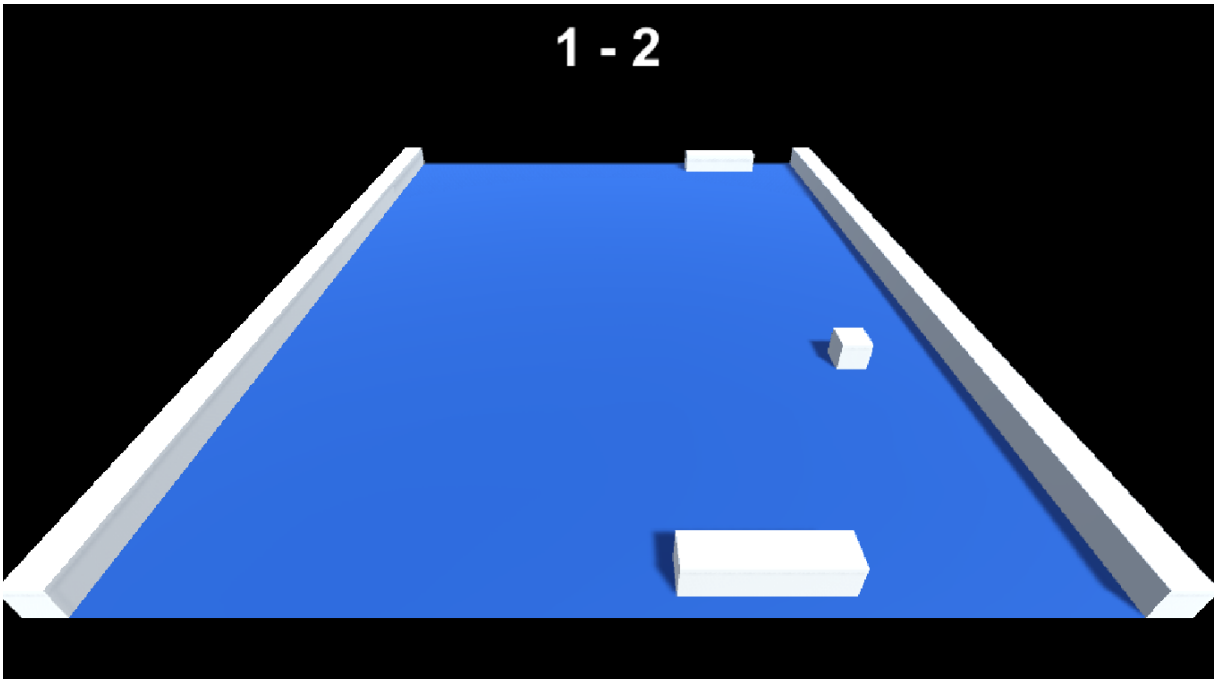
Pour le color grading, vous pouvez aussi paramétrer votre rendu. Cet effet permet de modifier l’image, ses couleurs, la saturation, le contraste, les nuances de couleur etc. Voilà un exemple de réglage :


Color Grading
All None
On
Off
Mode
High Definition Range
ColorSpace in project settings is set to Gamma, HDR color grading won't look correct. Switch to Linear or use LDR color grading mode instead.
Tonemapping
Mode
ACES
White Balance
Temperature
0
Tint
0
Tone
Post-exposure (EV)
0.96
Color Filter
HDR
Hue Shift
0
Saturation
35
Contrast
10
Channel Mixer
Red
Green
Blue
Red
100
Green
0
Blue
0
Trackballs
0,00 0,00 0,00
1,00 1,00 1,00
1,00 1,00 1,00
Lift
Gamma
Gain
Grading Curves
Hue Vs Hue
Override
Reset
(Not Overriding)

Tout cela nous donne le rendu final suivant :



Pour rappel et pour comparer, je vous partage de nouveau l'image sans les effets :

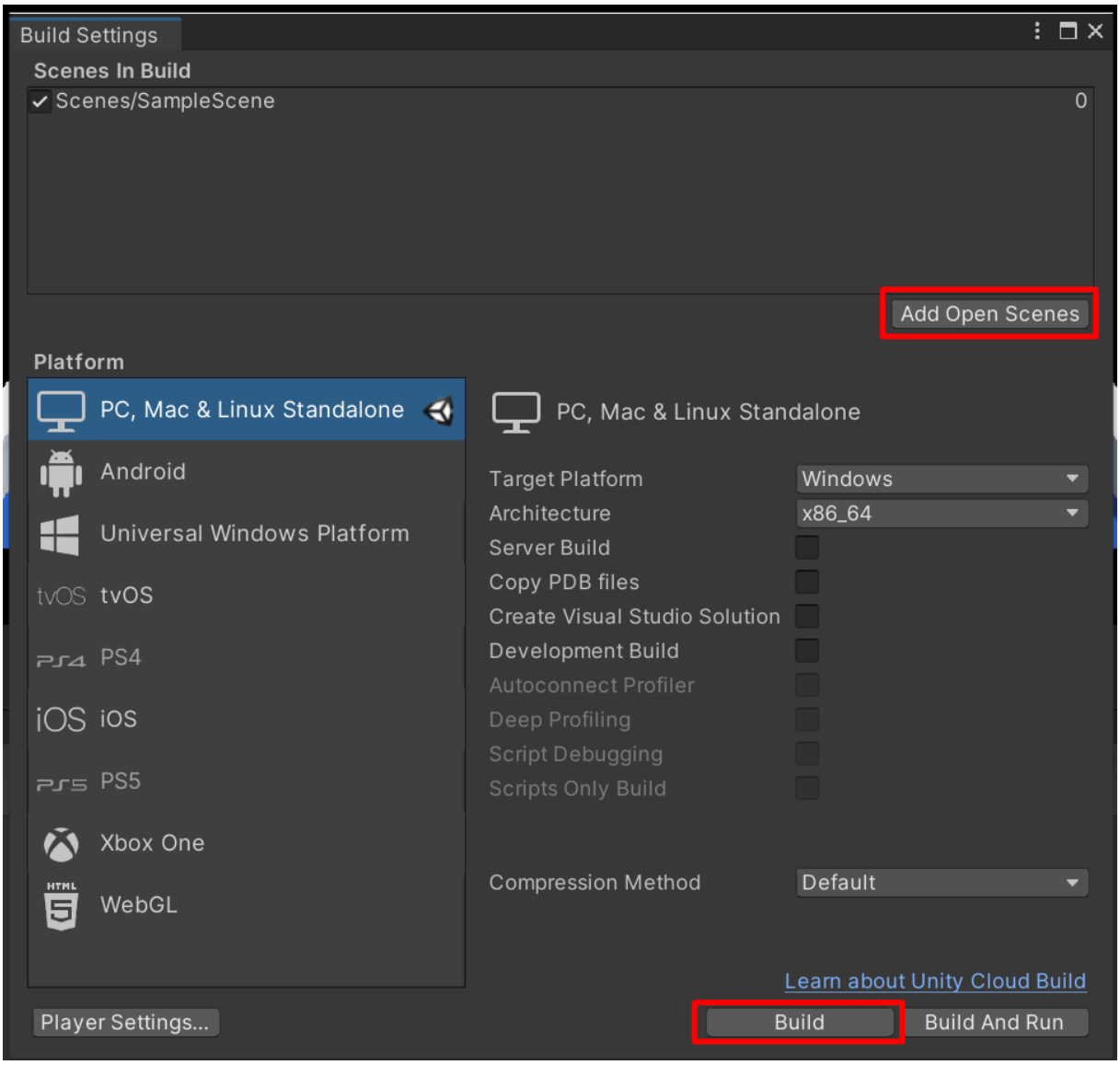


Vous remarquez que ces effets de post-traitement ont un impact majeur sur le rendu de votre projet.

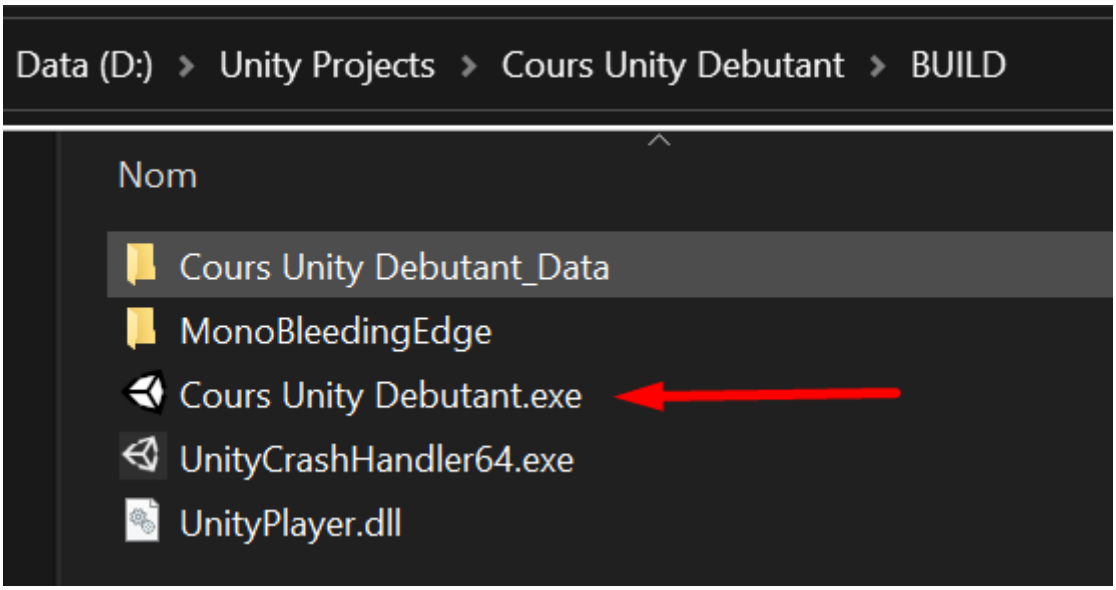
Notre jeu Pong est bien sûr un exemple basique et ultra simple avec à peine 6 modèles 3D au total mais vous avez déjà une bonne idée de l'impact.

Pour une idée plus précise sur le rendu d'une scène plus complexe, voici un comparatif avec / sans post-traitement :

Vous savez maintenant comment peaufiner votre projet avant de le partager. Et en parlant de partager, pour générer votre projet cliquez sur « File / Build Settings » puis cliquez sur « Add open Scenes » pour ajouter votre scène au build. Le build permet de générer l'exécutable de votre jeu afin de le partager. Cliquez ensuite sur « Build » pour lancer une compilation du projet :



Vous verrez qu'après compilation vous aurez votre exécutable avec ses dépendances :



Comme nous n'avons pas codé de menu, pour quitter le jeu il faudra faire un Alt + F4 (Sous Windows). Mais vous pouvez aussi modifier un script pour tester la touche échap et faire un Application.Quit() pour coder une fonction proprement. Un exemple complet est disponible ici : https://docs.unity3d.com/ScriptReference/Application.Quit.html

Voilà ce qui permet de conclure ce tutoriel Unity. Vous aurez vu les bases du développement sous Unity au travers d'un projet simple. Vous pouvez maintenant vous lancer sur des projets un peu plus complexes plus sereinement.

Pour aller plus loin

Découvrez la formation vidéo la plus complète au monde avec 170 heures de contenu et plus de 20 projets.

Un seul lien => https://www.udemy.com/course/formation-unity-par-la-pratique-le-cours-ultime-tout-en-1-unity/?referralCode=F158E1E111F89AEEAB52

C'est la formation qu'il vous faut pour maitriser Unity de A à Z et pour apprendre à créer n'importe quel type de jeux.

https://www.formation-facile.fr/