

## Programmation avec Unity

Ce tutoriel s'adresse aux personnes qui veulent commencer à apprendre à programmer avec Unity, mais qui ne savent pas par où débiter. Le but de ce tutoriel est de vous apprendre toutes les bases nécessaires pour faire de la bonne programmation avec Unity. Attention toutefois, je considère que vous avez de bonnes notions sur la programmation en C#.

Aussi, je considère que vous savez ce que signifient les termes Rigidbody, box/sphère/capsule collider, mesh, mesh renderer...

### Notion de base :

Dans Unity, les scripts sont des éléments (composent) que vous allez pouvoir attacher à vos objets de jeu (gameObject) pour qu'il applique la logique de jeu. Un script de base ne possède que deux fonctions : Start() et Update().

```
Test.cs
UnityVS.Unity.CSharp
- Test
- Start()

using UnityEngine;
using System.Collections;

public class Test : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }

}
```

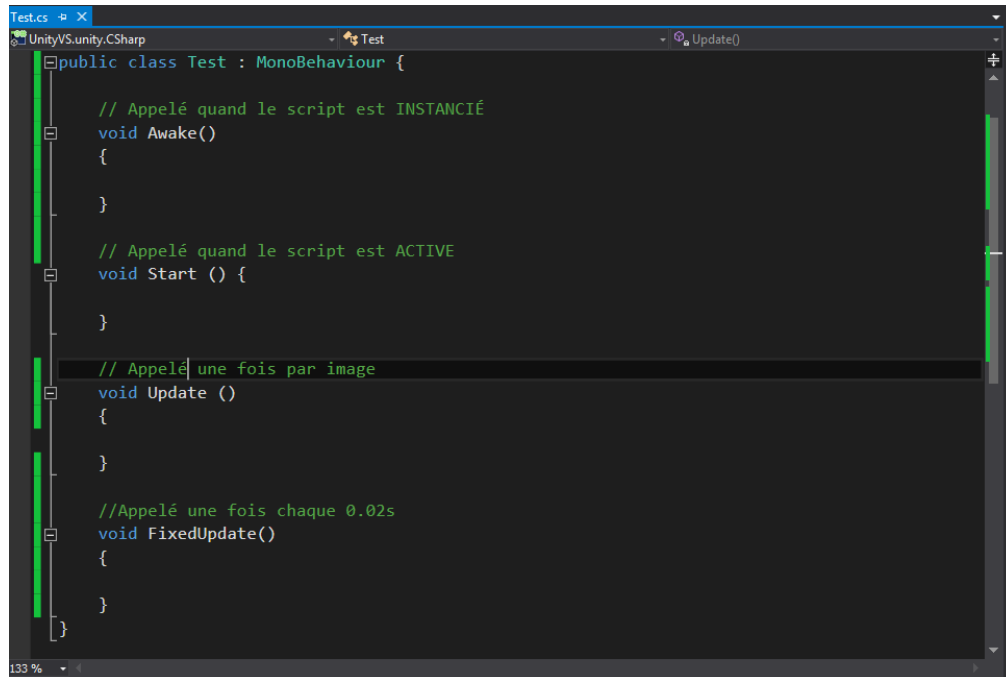
*N. B. Les photos prises sont faites sur l'IDE Visual Studio 2013. Si vous voulez changer de MonoDevelop à un autre IDE, allez dans Edit -> Preferences->External Tools et changez pour l'IDE que vous voulez.*

La fonction Start() sert en fait à initialiser ... ce que vous voulez ! Elle est appelée quand le script **est activé**. Cette partie est vraiment importante à retenir, car il existe des fonctions qui sont appelées quand le script **est instancié**. Une différence nette doit être faite.

La deuxième fonction est Update(). Cette fonction est appelée à chaque frame chaque fois qu'une nouvelle image est faite). Si vous êtes à 60fps, alors la fonction sera appelée 60 fois.

Il faut noter que dans un script vous n'avez pas le droit de créer un constructeur si vous héritez de MonoBehaviour. Vous **devez** passer par les fonctions d'instanciation qui vous sont offertes.

Améliorons un peu le script pour avoir une idée générale d'un script :



```
Test.cs
Unity/VS.unity.CSharp
Test
Update()

public class Test : MonoBehaviour {

    // Appelé quand le script est INSTANCIÉ
    void Awake()
    {

    }

    // Appelé quand le script est ACTIVE
    void Start () {

    }

    // Appelé une fois par image
    void Update ()
    {

    }

    //Appelé une fois chaque 0.02s
    void FixedUpdate()
    {

    }

}
```

Deux nouvelles fonctions sont apparues. Awake() et FixedUpdate().

La fonction Awake() est appelée quand l'objet est instancié. Utile si vous savez qu'un autre script va instancier votre script sans l'activer. Vous pouvez préparer votre script pour qu'il soit prêt à son lancement. Au départ, cette fonction n'a pas trop l'air utile, je vous l'accorde, mais il y a des cas où elle sera cruciale.

La fonction FixedUpdate() est appelée toutes les 0.02s. **TOUT CALCUL RELATIF À LA PHYSIQUE DOIT SE FAIRE ICI.** J'insiste, car vous ne voulez absolument pas que les calculs de physique soit dépendant du nombre d'images par seconde, sinon chaque machine aura un rendu de physique différent !

Bon, on a notre base ! On va maintenant passer à l'étape suivante. On va récupérer les variables qui nous intéressent afin de faire bouger un cube sans trop de soucis.

## Récupérer des attributs :

Je suis sûr que vous vous êtes déjà demandé comment récupérer mes éléments dans un script. Il existe plusieurs manières que je vais détailler ici afin d'initialiser vos variables. Pour cette partie, nous allons nous intéresser à deux aspects :

1. Récupérer un élément sur notre objet (l'objet sur lequel se trouve le script).
2. Récupérer un élément sur un autre objet.

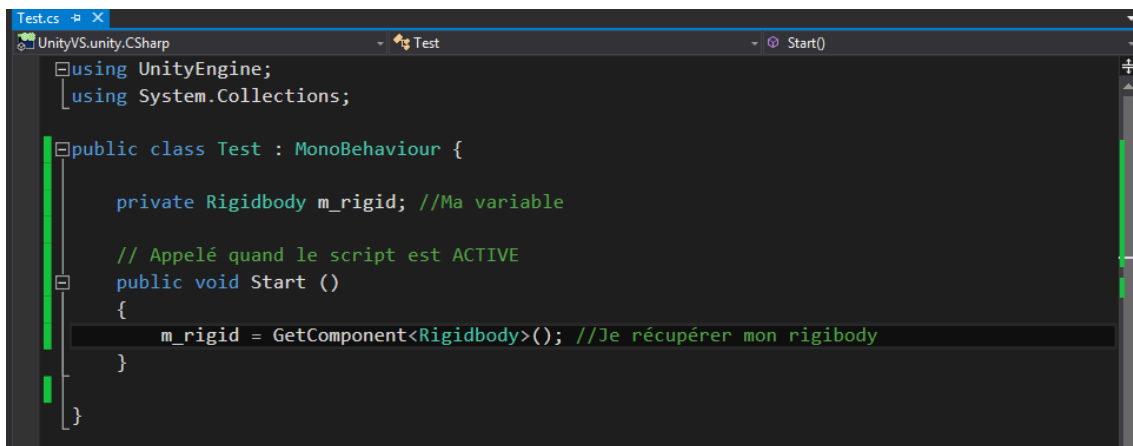
Commençons par le point 1. Pour récupérer un élément, on peut utiliser `gameObject.ElementEnQuestion`. Si vous utilisez Unity 5 comme moi, cette méthode est déconseillée. Il est préférable d'utiliser :

```
GetComponent<Component>();
```

Cette méthode sera votre dieu à partir de maintenant jusqu'à ce que vous arrêtez d'utiliser Unity. Elle vous permet de récupérer un élément à partir de votre objet. Vous pouvez lire ça comme :

```
this.gameObject.GetComponent<Component>();
```

Le **Component** constitue l'élément que vous voulez récupérer. Il suffit de le remplacer par l'élément pour l'obtenir. Exemple concret :



```
Test.cs
using UnityEngine;
using System.Collections;

public class Test : MonoBehaviour {

    private Rigidbody m_rigid; //Ma variable

    // Appelé quand le script est ACTIVE
    public void Start ()
    {
        m_rigid = GetComponent<Rigidbody>(); //Je récupérer mon rigidbody
    }

}
```

*NB : vous ne trouverez que le minimum dans les exemples fournis pour une meilleure visibilité.*

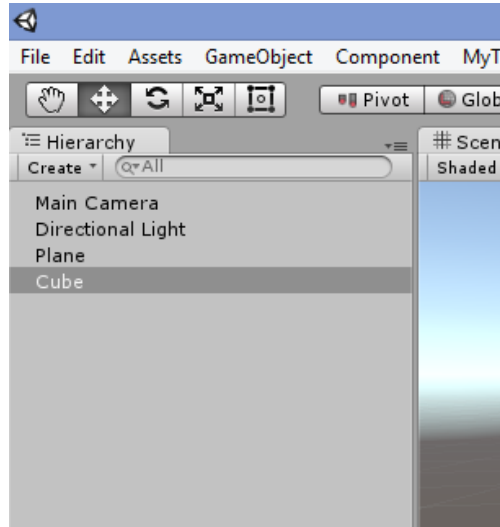
Facile, non ? Vous pouvez maintenant récupérer n'importe quel élément de votre objet avec cette méthode. Pour savoir quel élément il y a, regarder dans votre inspecteur. Si vous ne possédez pas l'élément, alors une erreur va apparaître avec **MissingComponentException** en précisant quel élément il vous manque. Assez sympa comme erreur.

Au fait, **vos scripts sont considérés comme des Component**. Si vous voulez faire des appels de méthodes dessus, il faudra récupérer le script qui vous intéresse puis appeler la méthode. (NB : n'oubliez pas, impossible d'appeler les méthodes non public... je dis ça comme ça).

Le point 2 est plus délicat, car il y a plusieurs manières de faire. Commençons par la plus simple :

```
GameObject.Find(« Nom de l'objet »).GetComponent<Component>();
```

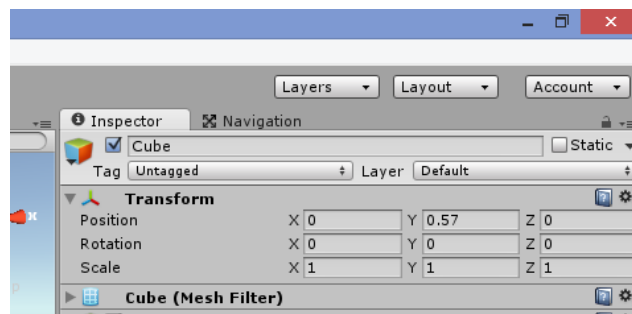
La méthode Find permet de retrouver n'importe quel objet du moment qu'il a le bon nom fourni en paramètre. Ce nom doit être le même que celui dans votre hiérarchie d'objet.



Ensuite, vous n'avez plus qu'à obtenir l'élément avec la méthode GetComponent. Le problème avec cette méthode c'est que vous pouvez avoir deux noms exactement similaires dans votre hiérarchie. Dans ces cas-là, Unity va retourner le premier qu'il trouve de haut en bas. Mais ne jouons pas au plus malin avec l'ordinateur, il va finir par gagner. Alors je vous propose d'utiliser une autre méthode :

```
GameObject.FindGameObjectWithTag(« Nom du tag »).GetComponent<Component>();
```

La méthode FindGameObjectWithTag va vous retourner un gameObject qui respecte le bon tag que vous fournissez en paramètre. Vous pouvez définir les tags ici :



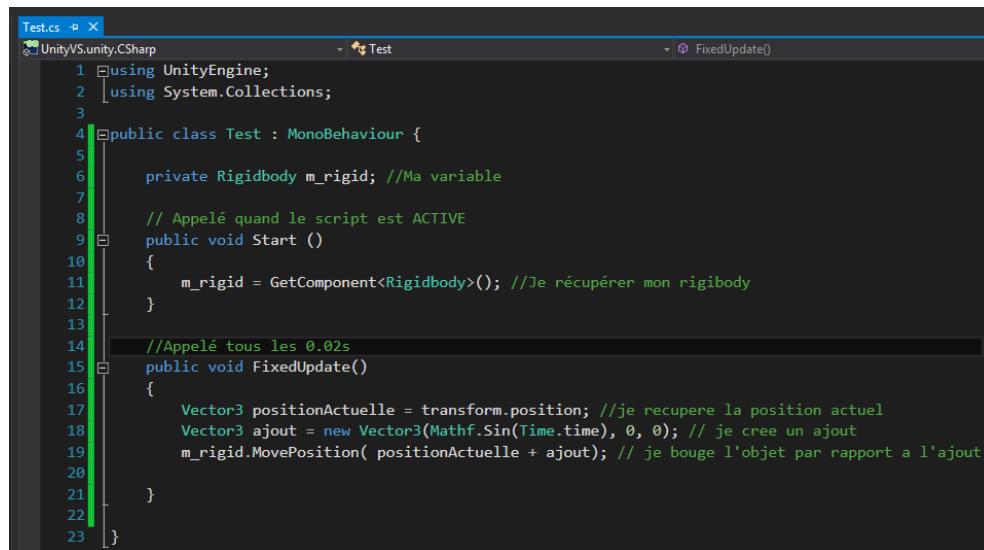
Ici, on peut voir que mon objet n'a pas de tag (Untagged). Si je veux, je peux créer un tag et le lui assigner. Cela peut être utile pour différencier un monstre, un joueur et une voiture.

Le reste de la méthode vous la connaissez déjà par cœur à présent !

La dernière méthode est très spécifique, mais utile. Si vous avez des enfants pour votre gameObject, la méthode GetComponentInChildren existe. Je vous laisse vous informer dessus par vous-même.

## Faisons bouger notre objet

Bon c'est mignon tout ce texte, mais si on faisait un petit quelque chose ? On va faire bouger un objet de gauche à droite. Attention, on accélère la cadence !



```
1 using UnityEngine;
2 using System.Collections;
3
4 public class Test : MonoBehaviour {
5
6     private Rigidbody m_rigid; //Ma variable
7
8     // Appelé quand le script est ACTIVE
9     public void Start ()
10    {
11        m_rigid = GetComponent<Rigidbody>(); //Je récupérer mon rigidbody
12    }
13
14    //Appelé tous les 0.02s
15    public void FixedUpdate()
16    {
17        Vector3 positionActuelle = transform.position; //je recupere la position actuel
18        Vector3 ajout = new Vector3(Mathf.Sin(Time.time), 0, 0); // je cree un ajout
19        m_rigid.MovePosition( positionActuelle + ajout); // je bouge l'objet par rapport a l'ajout
20    }
21
22 }
23 }
```

Dans la fonction Start(), on initialise notre variable m\_rigid en récupérant le Rigidbody de notre objet. Ensuite les choses intéressantes se passent dans FixedUpdate().

1. On récupère notre position. Transform est l'élément que possède tout gameObject sur Unity. Il définit la position, rotation et le scaling dans l'espace. On récupère notre position\* qui est sous la forme d'un Vector3 (vecteur à trois dimensions). Il existe aussi des Vector2 si vous faites de la 2D.
2. On va se créer un « ajout » que l'on additionnera à notre position pour faire un mouvement. Pour l'amusement, j'utilise la fonction sinus qui va osciller sur 1,-1 avec le temps écoulé comme paramètre (Time.time\*\*) sur l'axe x.
3. La fonction MovePosition est appelée sur le Rigidbody et on lui donne en paramètre le vecteur résultant.

TADA !! Notre premier script est en place et il fonctionne. Testez-le ! Le cube ira vers la droite pendant un temps puis reviendra à sa position originale. Facile Unity ! (N'oubliez pas de rajouter le script à un cube ou à un objet quelconque de test).

Si vous compilez ce code, vous remarquerez que l'objet accélère et c'est normal. La fonction sinus va graduellement monter vers 1 ou -1. Les ajouts seront de plus en plus conséquents donnant cette sensation (0, 0.1, 0.2, 0.3...).

Vous remarquerez sans aucun doute, si vous avez l'œil fin que le mouvement n'est pas « smooth » comme on dit. Il a l'air saccadé. Pour arranger ce problème, il faut multiplier l'ajout par `Time.fixedDeltaTime`. Si vous êtes dans un la fonction update, les opérations demanderont parfois un `Time.deltaTime`. Cela aura pour but de rendre les transitions plus agréables à l'œil. Pour mieux le voir, mettez une valeur brute comme 1 au lieu de la fonction sinus avec et sans `Time.fixedDeltaTime`.

*\* : on aurait pu utiliser la fonction `GetComponent` au lieu de `transform.position`. Le standard pour Unity 5 autorise parfaitement d'utiliser `transform`, c'est pourquoi je le fais.*

*\*\* : il existe la classe `Time`, qui permet de faire un grand nombre de choses par rapport au temps. Ici la fonction `Time.time` donne le temps total qui s'est écoulé depuis que l'application fonctionne (lorsque l'on appuie sur Play ou quand on lance l'exécutable du jeu fini).*

## Fonction utile : bouger, tourner et sauter

Dans cette partie vous ne trouverez pas d'exemple de code, mais une aide pour faire bouger, sauter et tourner vos personnages.

- **`transform.Translate(Vector3)`** : cette fonction permet de faire bouger un objet. C'est recommandé pour les objets qui font toujours le même mouvement et qui ne possèdent pas de RigidBody (exemple, une plateforme qui bouge sur laquelle sauter).
- **`transform.LookAt(Vector3)`** : cette fonction vous permet de tourner dans la direction du Vector3. Pareil, à ne pas utiliser sur des personnages ou ennemis, mais des objets inanimés.
- **`rigidbody.MovePosition(Vector3)`** : vous connaissez maintenant la fonction qui permet de déplacer un rigidbody selon un Vector3.
- **`rigidbody.MoveRotation(Vector3)`** : vous permet de tourner selon un Vector3.
- **`rigidbody.AddForce`** : vous permet d'ajouter une force dans une direction et selon un type d'impulsion. Pratique pour les sauts !

Vous trouverez plus d'informations sur le RigidBody et Transform sur la documentation officielle :

<http://docs.unity3d.com/ScriptReference/Transform.html>

<http://docs.unity3d.com/ScriptReference/Rigidbody.html>

## Instancier des objets :

On sait faire des scripts, on a compris comment faire bouger un objet, il faut maintenant savoir comment faire apparaître des objets sur scène. Toute cette partie est concentrée dessus et sur les différentes manières.

La première manière est d'appeler la méthode :

```
Instantiate(GameObject, position, rotation);
```

Ceci va créer un objet dans le monde avec une position et une rotation tel que fourni. Si vous omettez les deux derniers paramètres, l'objet apparaîtra en position 0x, 0y, 0z.

Maintenant, le problème ne consiste pas à appeler cette fonction (quoique, voir plus bas), mais à avoir une manière d'instancier le bon objet.

Ce que je veux dire par là, c'est que vous n'allez pas instancier un cube puis rajouter tout ce qui lui manque à coup de ligne de code ! Il serait plus simple de créer un objet qui possède déjà tout. Pour cela, il existe les prefabs.

<http://docs.unity3d.com/Manual/Prefabs.html>

Pour faire simple, c'est comme si vous aviez instancié un objet qui a déjà tout ce qu'il lui faut pour débiter son travail.

Je vous conseille de vous créer un dossier « **Ressources** » dans le dossier Asset avec exactement la même écriture. Il est possible de récupérer des objets/textures/son... depuis ce dossier à partir de script, ce qui n'est pas vraiment possible avec un autre nom. Pour récupérer un gameObject ou un prefab dans notre cas, il faudrait faire :

```
Resources.Load(« chemin/chemin/obj ») as GameObject;
```

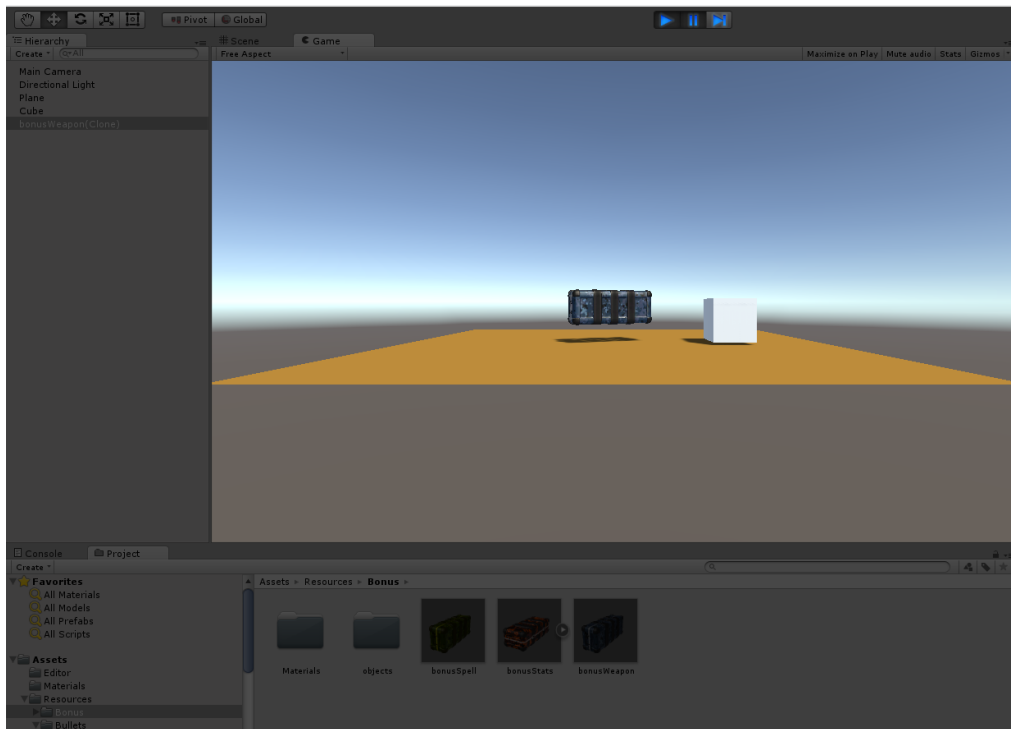
Il faut noter que la racine est Resources. Donc si on veut faire instancier un bonus dans le dossier Resources/Bonus/bonusWeapon alors on fera :

```
Resources.Load(« Bonus/bonusWeapon ») as GameObject;
```

Il ne faut pas oublier la conversion vers un GameObject, car la fonction Load() retourne un objet et non un GameObject, car elle ne sait pas ce que vous êtes en train de charger en mémoire. Exemple de code avec :

```
Test.cs
1 using UnityEngine;
2 using System.Collections;
3
4 public class Test : MonoBehaviour {
5
6     private bool isBorn; // Mon booléen de condition
7     private GameObject obj; // Mon objet a instancié sur la scene
8
9     // Appelé quand le script est ACTIVE
10    public void Start ()
11    {
12        obj = Resources.Load("Bonus/bonusWeapon") as GameObject; //L'objet que je veux dupliquer
13
14        isBorn = false;
15    }
16
17    //Appelé toutes les "frames"
18    public void Update()
19    {
20        if (isBorn) //si c'est fait, on retourne
21            return;
22        else //sinon, on fait le nouveau obj
23        {
24            isBorn = true;
25            Instantiate(obj); //on créer l'objet
26        }
27    }
28 }
```

Le résultat est le suivant :



*N. B. Il est possible de changer les couleurs pour savoir si vous êtes en mode Play ou non. C'est très utile pour ne pas perdre son travail, car il n'y a pas de sauvegarde faite en mode Play. Il faut aller dans Edit->Preferences->Color->PlayMode tint*

Assez simple non ?

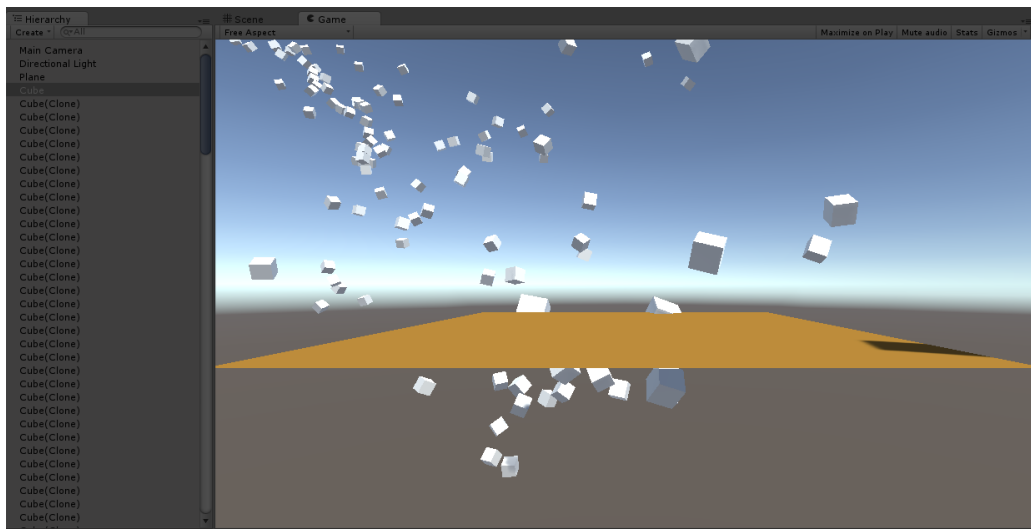


Une idée simple serait de faire un cube qui tire des boules toutes les deux secondes. Un pistolet !

Il est possible de mal utiliser la fonction Instantiate(). Regardez cet exemple :

```
Test.cs
UnityVS.Unity.CSharp
1 using UnityEngine;
2 using System.Collections;
3
4 public class Test : MonoBehaviour {
5
6     private bool isBorn; // Mon booléen de condition
7     private GameObject obj; // Mon objet a instancié sur la scene
8
9     // Appelé quand le script est ACTIVE
10    public void Start ()
11    {
12        obj = GameObject.Find("Cube"); // L'objet que je veux dupliquer
13
14        isBorn = false;
15    }
16
17    // Appelé toutes les "frames"
18    public void Update()
19    {
20        if (isBorn) // si c'est fait, on retourne
21            return;
22        else // sinon, on fait le nouveau obj
23        {
24            isBorn = true;
25            Instantiate(obj); // on créer l'objet
26        }
27    }
28}
```

Tout à l'air correct non ? On trouve notre objet du nom de cube sur la scène, on l'instancie une fois et plus jamais on ne le fera de nouveau... si seulement vous aviez raison :



*NB : faire une pluie de cube, c'est sympa, mais ce n'est pas attendu...*

Mon dieu, que se passe-t-il donc ? En fait, c'est assez simple. Mon 1<sup>er</sup> cube crée un nouveau cube **exactement identique** à la seule différence que les scripts du nouveau cube **commencent leur début de vie** (donc appel à Awake() puis Start()).

La variable `isBorn` est fausse sur le nouveau cube donc, il va en instancier un nouveau et le nouveau ben... il va faire pareil. Une boucle sans fin est faite.

Cet exemple est juste pour vous montrer qu'il faut quand même faire attention à ce que vous faites dans Unity, surtout dans les méthodes `Update`. Des problèmes similaires, mais plus sournois peuvent arriver.

#### Ressources externes :

C'est déjà la fin. Vous savez maintenant ce qu'un script. Vous êtes capable de déplacer des objets et de les faire pivoter (tentez-vous, c'est simple et marrant). Vous pouvez instancier des objets et récupérer des éléments sans aucun souci. Mais je ne vous laisse pas sans rien, voici une liste de référence vers différentes parties du document officiel :

Transform : <http://docs.unity3d.com/ScriptReference/Transform.html>

Rigidbody : <http://docs.unity3d.com/ScriptReference/Rigidbody.html>

MonoBehaviour : <http://docs.unity3d.com/ScriptReference/MonoBehaviour.html>

Ressources : <http://docs.unity3d.com/ScriptReference/Resources.html>

GameObject : <http://docs.unity3d.com/ScriptReference/GameObject.html>

Vector3 : <http://docs.unity3d.com/ScriptReference/Vector3.html>

Time : <http://docs.unity3d.com/ScriptReference/Time.html>

Et je vous laisse des bonus si vous voulez allez plus loin :

Input : <http://docs.unity3d.com/ScriptReference/Input.html> (entrée clavier)

NavMesh : <http://docs.unity3d.com/ScriptReference/NavMesh.html> (IA)

Collider : <http://docs.unity3d.com/ScriptReference/Collider.html> (Physique)

Collision : <http://docs.unity3d.com/ScriptReference/Collision.html> (Physique)

Debug : <http://docs.unity3d.com/ScriptReference/Debug.html> (Bogue)

Ainsi que les deux liens directs vers le manuel et l'API Unity :

API : <http://docs.unity3d.com/ScriptReference/index.html>

Manuel : <http://docs.unity3d.com/Manual/ScriptingSection.html>