

TD2 : Gestion de la physique et fonctions de mise à jour

Florian Jeanne

Rémy Frenoy

Yann Soullard

Solution du TD

Retour sur le TD 1

Au cours du TD1, vous avez appris à créer un *GameObject* simple comme une sphère et à la déplacer à l'aide de la méthode `AddForce()`. Dans ce deuxième TD, nous allons étudier comment fonctionne la physique dans Unity, et notamment les collisions entre objets.

Update - FixedUpdate - LateUpdate

Si vous reprenez votre code, dans le script `BallController`, les `AddForce()` étaient appelées dans la méthode `KeyboardMovements()`, qui était elle-même appelée dans la méthode `Update()`. Cependant, ce n'est pas tout à fait correct. Mais avant d'aller plus loin, nous allons faire un petit rappel de cours.



Une *frame* est une image projetée à un instant donné et pour une certaine durée. Au cinéma par exemple, les films sont projetés avec une fréquence de 24 images par secondes, ou 24

frames per second en Anglais. Dans un projet Unity, une frame est donc une capture de votre scène à un instant donné.

Update() est appelée juste avant le rendu d'une *frame*. Aussi, si le temps de calcul pour une *frame* est plus long que celui de la *frame* suivante, l'intervalle de temps entre les appels de la méthode Update() sera différent.

FixedUpdate() quand à elle, est appelée à intervalles de temps réguliers, indépendamment du rendu de la *frame*. Cette méthode est appelée juste avant les calculs liés à la physique. Vous voyez où je veux en venir ? En effet, c'est bien dans cette méthode qu'il va falloir appeler nos méthodes agissant sur la physique des GameObjects comme AddForce().

Reprenez maintenant votre script BallController, et appelez KeyboardMovements() dans FixedUpdate() au lieu de Update().

```
15  void Update ()
16 | {
17 |
18 | }
19 |
20  void FixedUpdate () {
21 |     KeyboardMovements ();
22 | }
```

La question se pose également pour la gestion de notre caméra. En effet, on modifie la position de la caméra à chaque frame en fonction de la position de notre balle. Il existe une méthode pour cela, LateUpdate().

LateUpdate() est utilisée pour des caméras de suivi, l'animation procédurale, ou lorsque l'on souhaite connaître le(s) dernier(s) état(s) d'un objet. Elle est également appelée à chaque *frame*, une fois que tous les éléments présents dans Update() ont été traités. Ainsi, dans notre situation, on est sûr que la balle a bien été déplacée avant d'utiliser sa position actuelle pour modifier celle de la caméra.

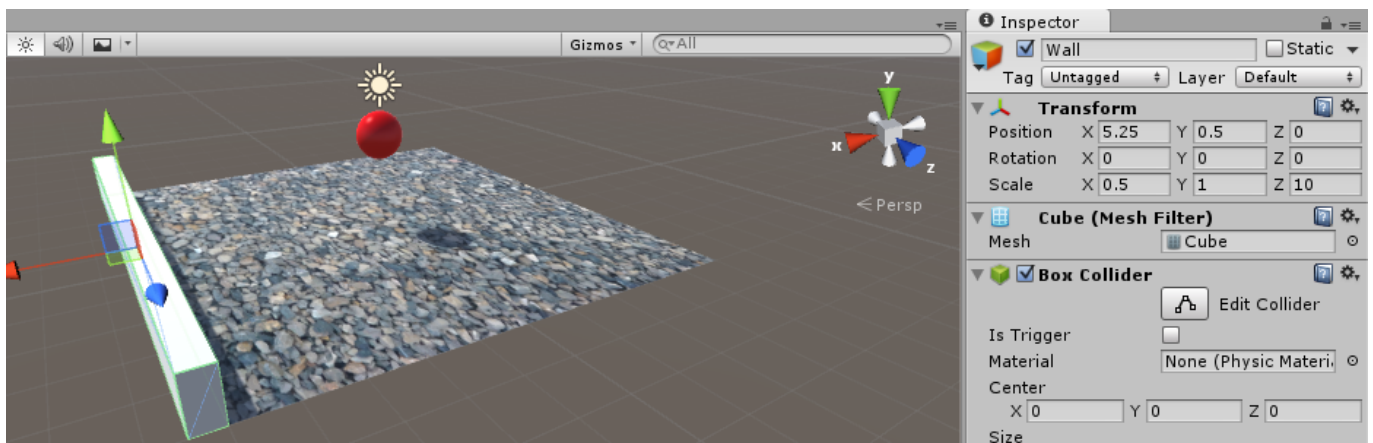
Ainsi, si nous récapitulons tout ça :

- Start() est appelée une seule fois à la première exécution du script qui le contient ;
- FixedUpdate() est appelée à intervalles réguliers avant le moindre calcul physique ;
- Update() est appelée avant le rendu de la *frame* ;
- LateUpdate() est appelée à la suite de Update().

Bien entendu, il existe beaucoup plus de méthodes événementielles dans Unity dont l'exécution est prédéterminée. Pour votre information : [Execution Order of Event Functions](#).

Prefabs

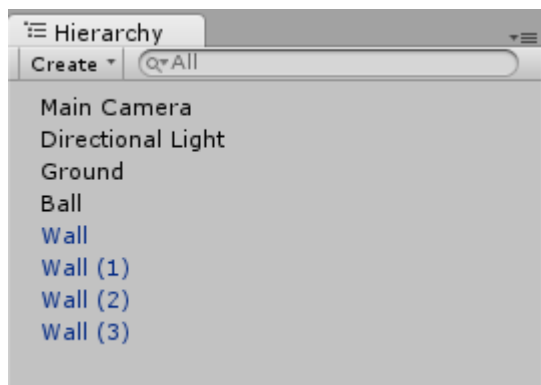
Ouvrez la scène de votre projet du TD1 en important le package que vous aviez créé. Nous allons créer des murs pour empêcher notre balle de tomber. Pour cela, créez un cube à partir des *GameObject* Unity (**GameObject > 3D Object > Cube**). Modifiez les dimensions et la position du cube pour qu'il agisse comme un mur sur l'un des bords du terrain "ground".



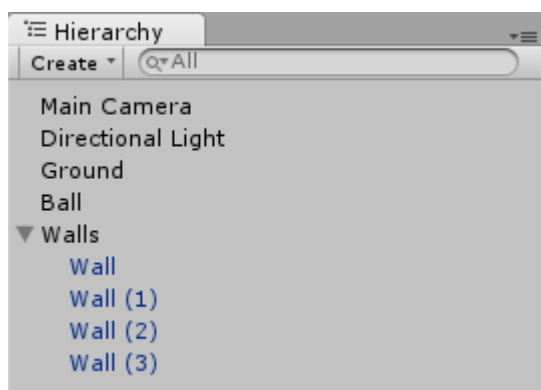
Il nous reste donc trois murs à faire. On pourrait tout simplement refaire la même manipulation trois fois de suite. Ou alors... On pourrait utiliser ce que l'on appelle un *prefab*. Un prefab est un objet Unity qui conserve un *GameObject* quelconque, avec ses propriétés et ses composants. Il agit comme un *template* (ou modèle) à partir duquel vous allez pouvoir créer différentes instances dans la scène. Ainsi, si vous modifiez par la suite votre prefab en changeant la valeur de l'une de ses propriétés, cela se répercutera sur toutes les instances présentes dans le projet.

Il existe plusieurs manières de créer votre prefab. Un simple glisser/déposer depuis votre *Hierarchy* vers la fenêtre du projet créera automatiquement un prefab, ou alors en effectuant un clic droit dans le projet, *Create, Prefab*. Votre prefab est cependant vide. Il vous faudra effectuer un glisser/déposer de votre *Hierarchy* vers le prefab.

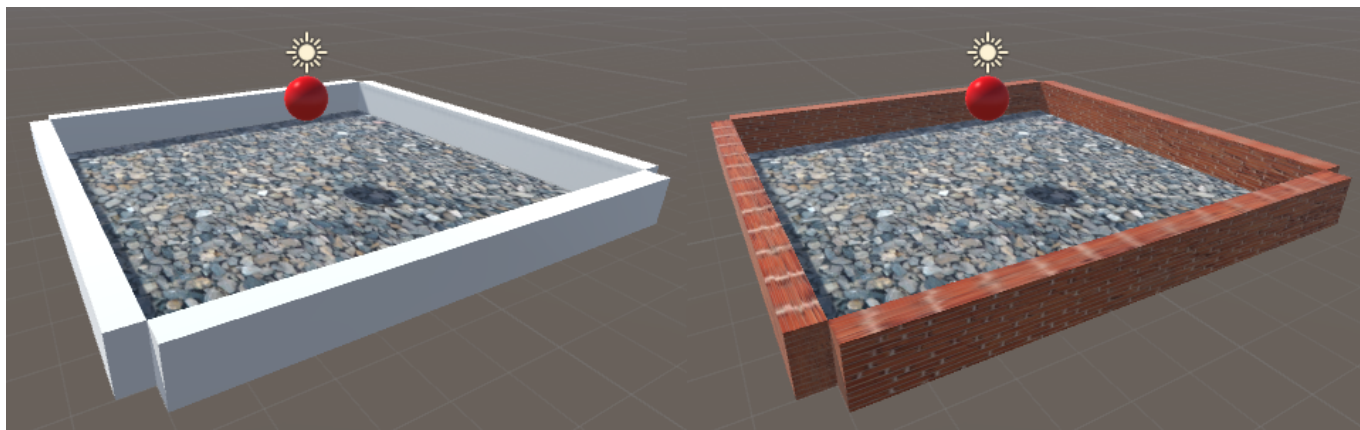
Créez donc un prefab de votre mur et générez trois instances de celui-ci par un glisser/déposer (encore et toujours) vers votre scène ou votre *Hierarchy*.



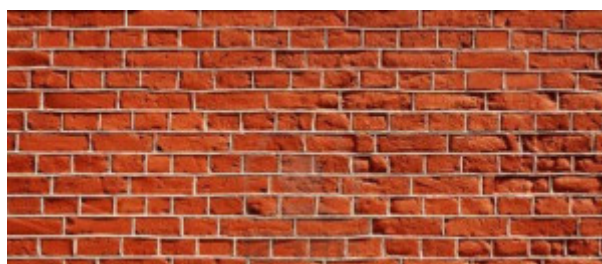
Wall (1) ? Wall (2) ? Mais qu'est-ce que... ? Il va falloir réorganiser tout ça ! Cependant, il n'existe pas de "dossier" dans la *Hierarchy* à proprement parler, on utilise donc des GameObject vide à la place. Pour cela, **GameObject > Create Empty**.



Enfin, disposez les nouveaux murs tout autour du terrain afin que notre balle ne puisse plus tomber. Vous pouvez également ajouter une texture sur votre prefab de mur, et elle s'appliquera automatiquement à toutes les instances présentes, comme ceci :



Vous pouvez télécharger une texture de briques ici :



Cliquez pour télécharger

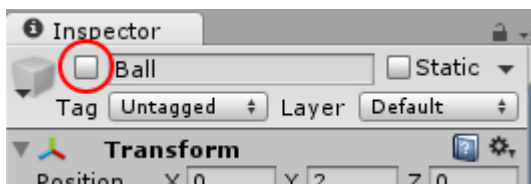
AddForce vs Translate

Jusqu'à présent, nous déplaçons notre balle à l'aide de la méthode `AddForce()`, qui s'applique sur le *rigidbody* de celle-ci. Cependant, il existe une autre méthode pour effectuer un déplacement. Celle-ci, au lieu d'appliquer des forces sur notre objet, modifie directement sa position en agissant sur son composant *transform*.

La méthode à appeler est `Translate()`. Elle s'applique sur un objet ne possédant pas de *rigidbody*. Créez donc un nouveau *GameObject*, un cube par exemple, auquel vous attachez un nouveau script *CubeController*. Dans ce script, vous créez une méthode `KeyboardMovements()`. Elle est assez similaire à celle de *BallController*, sauf qu'au lieu d'appeler des `AddForce()`, vous appelez ici des `Translate()` sur votre *transform* pour le déplacement, et éventuellement des `Rotate()` si vous voulez faire des rotations. Utilisez par

exemple les touches « Z », « Q », « S », « D » à la place des flèches directionnelles pour ne pas interférer avec le déplacement de la balle.

Une fois la méthode terminée, sauvegardez. Pour ne pas qu'il y ait des collisions entre la balle et le cube, désactivez la balle. Pour cela, il faut la sélectionner, et dans l'*inspector*, décocher la *checkbox* située à côté du nom du *GameObject*.



Désormais, votre balle n'apparaîtra plus dans la scène. Lancez donc l'application et déplacez-vous.

Les différences entre les deux méthodes sont assez significatives. Avec `AddForce()`, vous prenez en compte la physique de Unity. Ainsi, les collisions sont gérées et la gravité est prise en compte. Avec `Translate()`, vous modifiez la position de l'objet à chaque *frame*. Il y a donc pas réellement de déplacement. L'objet se "téléporte" à chaque *frame*.

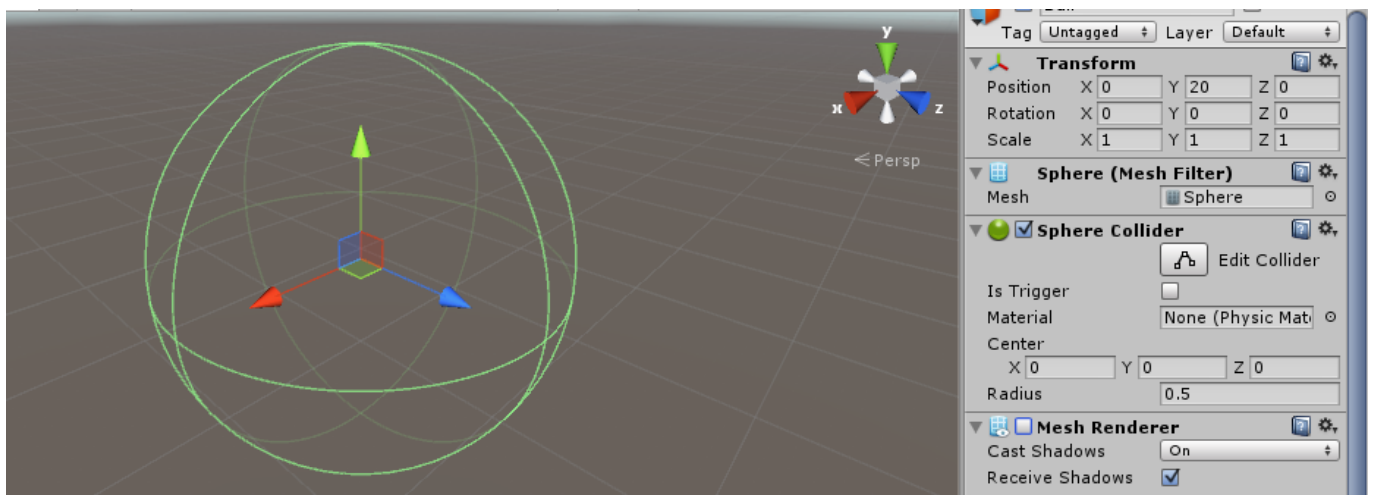
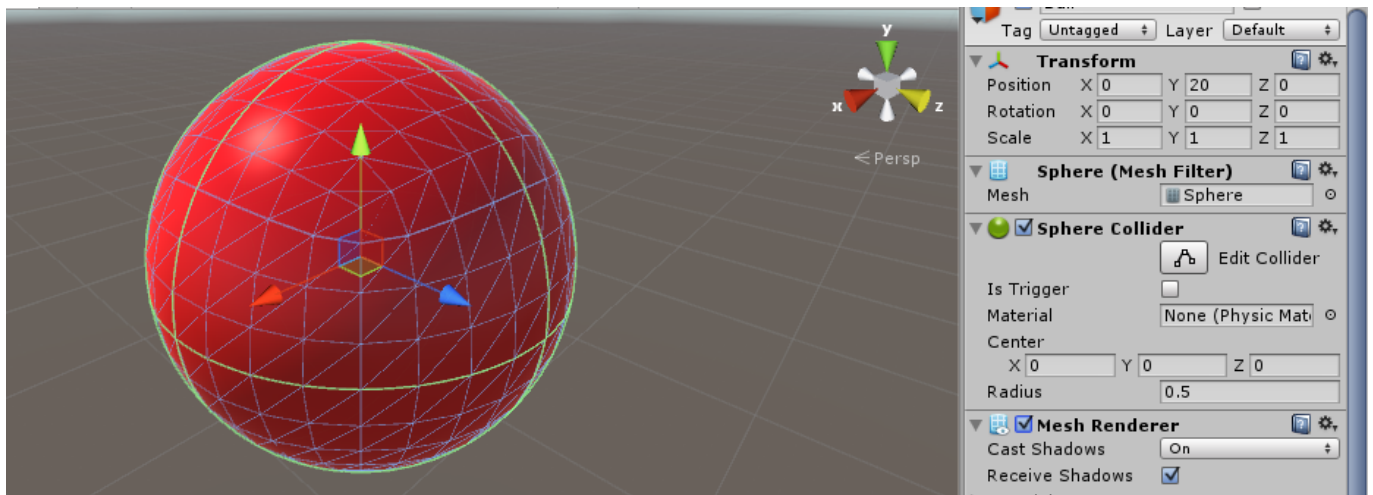
Collisions

Néanmoins, si vous ré-activez la balle, les collisions ont bien lieu entre celle-ci et le cube. Ceci est dû à la manière dont est gérée la physique dans Unity, et notamment les collisions.

Pour qu'une collision ait lieu entre deux objets, il faut qu'ils aient tous les deux des composants *Collider* et qu'au moins l'un d'entre eux possède un *Rigidbody*. Ainsi, si l'on reprend notre balle et notre cube, les collisions sont bien gérées entre la balle et les autres éléments de la scène, puisqu'elle possède un *Rigidbody* et que par défaut tous les *GameObjects* non-vides ont un *collider*. Notre cube cependant, passe au travers des murs puisque ni celui-ci, ni les murs n'ont de *Rigidbody*.

Les *colliders* sont représentés en vert dans votre scène lorsque vous sélectionnez un

gameObject. Ils peuvent avoir une forme primitive comme une **sphère**, une **box** ou une **capsule**, ou pour des formes plus complexes il existe deux méthodes. Prenons l'exemple d'une table. Si l'on prend un *BoxCollider* faisant la taille de la table, on ne pourra pas faire passer notre balle en dessous de celle-ci puisqu'elle entrera en collision avec son *collider*. Cependant, si l'on divise la table entre les quatre pieds et le plateau, et que l'on applique un *collider* adéquat à chaque partie, notre balle pourra passer en dessous. La deuxième solution consiste à appliquer un **MeshCollider** qui viendra épouser la forme définie.



De base, Unity gère les collisions entre deux objets de manière assez simple. Mais vous pouvez déclencher des actions spécifiques que vous avez au préalable définies. Par exemple,

le contact entre la balle et le cube de notre scène pourrait détruire l'un des deux objets. Reprenons notre script de la balle, `BallController`. Nous allons redéfinir la méthode `OnCollisionEnter()` qui est appelée lorsqu'il y a une collision. En C# :

```
void OnCollisionEnter(Collision collideEvent)
{
}
```

Ici, le paramètre `collideEvent` est de type `Collision` qui décrit une collision. Si vous regardez la documentation, vous verrez que l'on peut récupérer l'objet avec lequel notre *gameObject* entre en collision (`collideEvent.gameObject`). Pour détruire un objet, on utilise la méthode `Destroy()`. Cependant, on ne cherche pas à tout détruire. Il faut détruire uniquement le cube.

Ce type de fonction est appelé automatiquement en réponse à un événement spécial qui s'est déclenché dans votre scène (ici, une collision). Il n'y a pas besoin de l'appeler dans votre `Update()`.

Triggers

Les *Triggers* ressemblent très fortement aux collisions. En effet, les *Triggers* sont un type particulier de *Colliders*. Ils ne vont pas chercher à détecter les collisions, mais plus simplement détecter lorsqu'un *collider* entre dans l'espace d'un autre *collider* sans qu'il n'y ait de collision. Pour créer un *trigger*, il suffit de modifier la propriété *IsTrigger* d'un *collider* en modifiant sa valeur pour qu'elle soit égale à *true* (ou alors tout simplement cocher la case *IsTrigger* d'un *collider* dans l'*inspector*).

Comme il n'y a pas de collision dans un *Trigger*, c'est la méthode `OnTriggerEnter()` qui est appelée lorsqu'un *collider* pénètre dans l'espace.

Time.deltaTime

Maintenant que nos collisions sont gérées, revenons à notre déplacement. Vous avez sûrement remarqué que si l'on augmente un peu les valeurs de nos variables `tSpeed` (utilisée dans les translations) et `rSpeed` (utilisée dans les rotations), le cube se déplace très rapidement. La raison est simple. Le cube se déplace en fonction des appels d'`Update()`. Du coup, si vous avez un `tSpeed` égal à 2, le cube se déplace à raison de 2m par frame. Ce qui est assez rapide sachant que la durée d'une *frame* est en moyenne d'environ 17ms (16,666666666.. ms pour les plus puristes d'entre vous, et pour 60fps).

Pour connaître la durée d'une *frame*, il suffit d'afficher le paramètre `Time.deltaTime` dans la console grâce à la méthode `Debug.Log()` en l'appelant dans `Update()`. Si vous l'appellez dans `FixedUpdate()`, la valeur sera toujours égale à 0.02ms, puisqu'elle est appelée indépendamment du rendu des *frames*.

Mais quel est le rapport avec le déplacement du cube ? Sachant que le cube se déplace en mètres par frame, et que l'on connaît la durée d'une *frame*, si l'on multiplie notre variable dans les `translate()` et les `rotate()` par cette durée `Time.deltaTime`, alors le déplacement s'effectuera en mètres par seconde. Je m'explique. Par exemple, si `tSpeed = 2`, en le multipliant par `Time.deltaTime = 0.017`, avec l'appel de la méthode `Translate()` le cube se déplacera de :

$$2 * 0.017 = 0.034 \text{ mètres par frame}$$

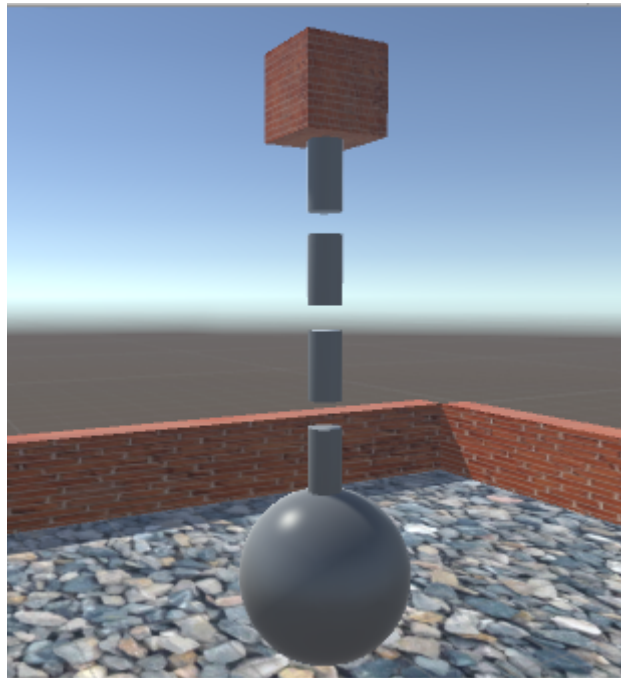
Ainsi, au bout d'une seconde (ou 1000ms) on aura $1 / 0.017 = 58.8$ soit 59 appels d'`Update()`. Du coup, votre cube se sera déplacé de $0.034 * 58.8 = 1.9992$ soit 2 mètres au bout de 1 seconde. La multiplication par `Time.deltaTime` nous permet donc bien de déterminer la vitesse en mètre par seconde. Aussi, visuellement les déplacements seront plus fluides car constants.

(Ce cours de mathématiques est sponsorisé par l'UV PS90)

Boule de démolition

Le cours c'est sympathique, mais la pratique c'est mieux. Après cette longue et fastidieuse partie de cours, nous allons enfin pouvoir tester toutes ces notions.

Nous allons créer une boule de démolition. Pour cela, nous allons utiliser une sphère qui sera attachée à l'aide d'une chaîne. Comment créer une chaîne ? C'est ce que nous allons voir tout de suite. Tout d'abord, supprimez le cube de votre scène, nous n'avons plus besoin.



*I came in like a wreeeeeking
baaaall...*

La boule de démolition est composée d'une sphère et de plusieurs (ici quatre) cylindres ou capsules (au choix), et d'un cube qui sert de socle. Commencez par positionner une sphère légèrement au dessus du plan, elle ne doit pas le toucher. Puis, créez un cylindre (ou une capsule évidemment) que vous positionnez de façon à ce qu'il soit légèrement enfoncé dans la sphère. Vous devrez bien évidemment modifier sa taille pour qu'il soit plus petit que la sphère. Enfin, ajoutez lui un *rigidBody*.

Ajoutez un composant *Fixed Joint* (**Component > Physics > Fixed Joint**) à votre sphère. Pour fonctionner, ce composant a besoin d'un autre *rigidBody*. Faites donc glisser le cylindre sur la variable pour connecter la sphère et le cylindre. Les deux objets sont maintenant liés, mais nous devons également créer les autres maillons de la chaîne.

Créez donc un second cylindre. Vous pouvez dupliquer le premier à l'aide du raccourci Ctrl + D. Cette fois-ci au lieu d'avoir un *Fixed Joint*, nous allons utiliser un *Hinge Joint* sur le premier cylindre. Faites glisser le second cylindre sur la variable correspondante comme précédemment. Répétez ces opérations plusieurs fois pour avoir par exemple quatre cylindre liés entre eux et une chaîne conséquente. Ajoutez un cube par exemple à la fin de la chaîne, avec un *rigidBody* et liez-le au dernier cylindre. Seulement, à cause du *rigidBody*, notre cube tombe à cause de son poids.. Néanmoins, on peut figer sa position et ses rotations dans les contraintes du *rigidBody*. Cochez donc toutes les cases.

Les *Fixed Joint* créent des liaisons fortes entre les objets (de la même manière que la hiérarchie). Les *Hinge Joint* permettent de créer des liaisons pivot (un seul degré de liberté).

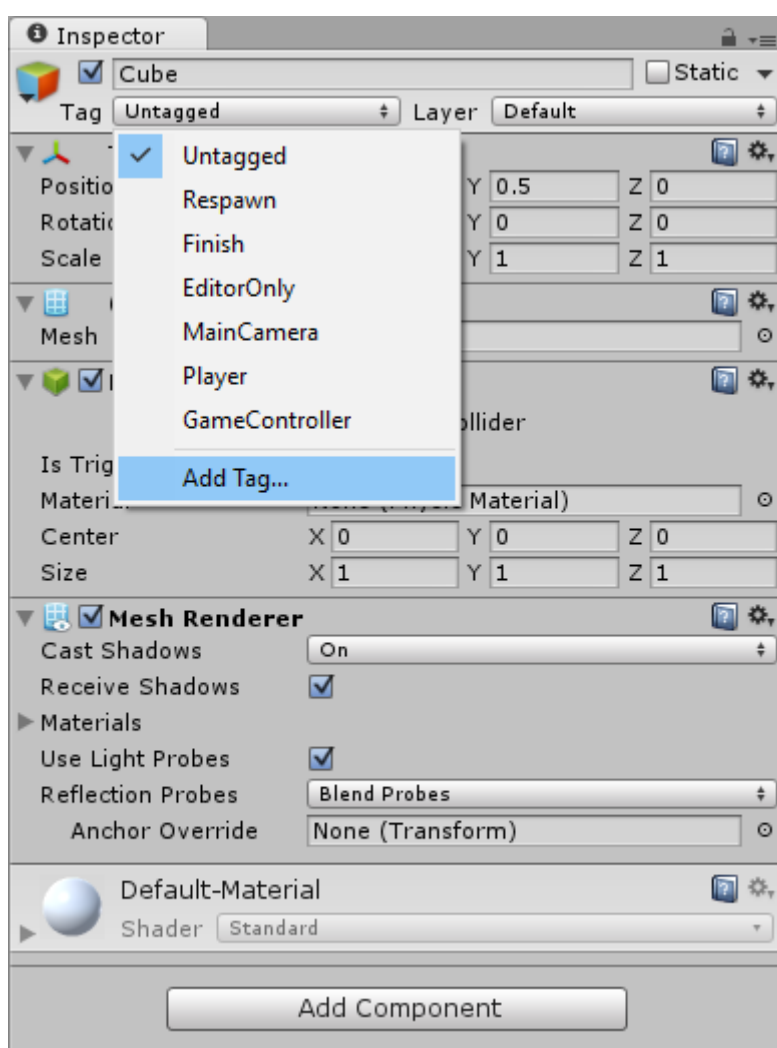
Votre boule de démolition est terminée. Seulement, elle ne bouge pas... On ne va pas démolir grand chose. Créez donc un nouveau script qui va nous servir à faire bouger la sphère et que l'on appellera *DemolitionBallController*. Cette fois-ci, on ne bougera pas la sphère manuellement. La sphère doit constamment recevoir une force pour qu'elle se déplace sans interruption. De plus, lorsqu'elle entre en collision avec notre balle, cette dernière doit retourner à sa position initiale avec une **vitesse** nulle. On pourrait par exemple, générer une force aléatoire à l'aide de la méthode **Random.Range()**.

Un tag « Target », ça en jette vachement plus

On souhaite désormais ajouter des cibles sur la scène que l'on devra "manger" avec notre boule, tout en évitant les boules de destruction.

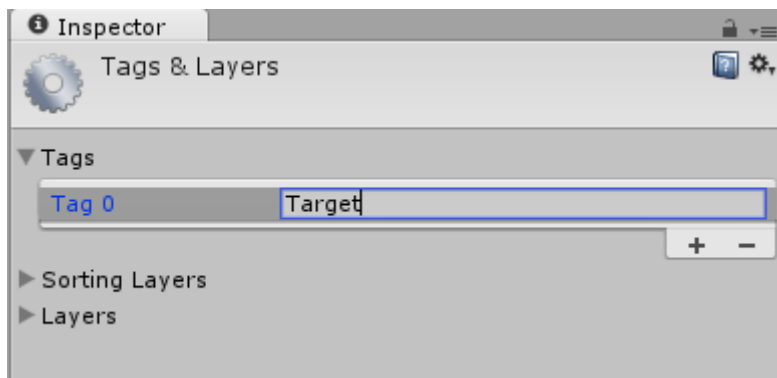
Commençons par ajouter un cube sur la scène. Puisqu'il a un rigidbody, et que le script de la boule détruit les objets nommés "Cube", notre cube disparaît bien lors de la collision. Il ne peut cependant y avoir qu'un objet nommé "Cube", alors que l'on souhaite avoir de nombreuses cibles à détruire. Plutôt que de détruire les objets nommés "Cube", nous allons créer un **tag** "Target", et nous détruirons tous les objets ayant ce tag.

Pour cela, sélectionnez le cube, puis sélectionnez *Add Tag* dans la liste déroulante *Tag*.



Ajout d'un nouveau tag

Il faut alors créer un nouveau tag, que nous appellerons "Target".



Création du tag

Vous pouvez désormais retourner dans l'inspector du Cube, et lui affecter le tag nouvellement créé.

Nous devons maintenant modifier le code de la balle (et plus précisément la méthode *onCollisionEnter*) afin qu'elle détruise non pas les objets nommés "Cube", mais les objets tagés "Target". Une petite lecture des fonctions publiques d'un **gameObject** devrait vous aider.

Une fois la méthode modifiée, lancez votre application, le cube devrait être détruit lors d'une collision avec la balle. Nous avons désormais la possibilité de détruire n'importe quel objet disposé sur la scène, à partir du moment où il possède le tag "Target". Plutôt cool, non ? Peut-être, mais pour l'instant nous n'avons qu'un pauvre cube à détruire...

Instanciation automatique des cubes

On souhaite avoir de nombreux cubes sur la scène, et surtout on souhaite que de nouveaux cubes apparaissent lorsque l'on en détruit (un vrai génocide de cube !).

Créez un prefab "Cube" à partir de l'objet Cube, puis supprimez l'objet Cube de la scène.

Créez un **gameObject** vide que l'on nommera "CubeFactory". Ce manager sera chargé de construire des cubes à des positions aléatoires sur la scène, tout en ne dépassant pas une limite de cubes que nous fixerons à 10.

Créez donc un script "CubeGenerator" et assignez-le à l'objet CubeFactory.

Dans ce script, nous souhaitons faire les choses suivantes :

- Au démarrage de l'application, le script doit récupérer la position et la taille du plan, ainsi que la taille du prefab Cube. Il est possible de récupérer le renderer d'un objet grâce à la méthode *GetComponent*.
- A chaque mise à jour, le script doit générer un cube s'il y en a moins de 10 sur la scène. Le cube doit être placé à une position aléatoire **sur le plan de la scène**.

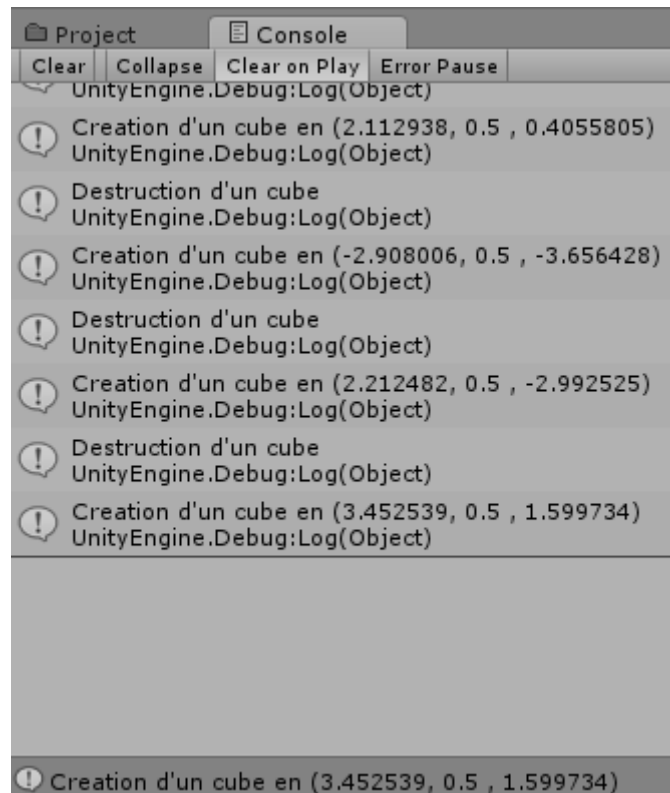
Informations utiles :

- Les informations concernant la taille et la position peuvent être retrouvées depuis les *bounds* du *renderer*. Le renderer d'un gameObjet étant récupérable en appelant la méthode *GetComponent* sur cet objet.

```
Renderer objectRenderer = object.GetComponent<Renderer>();
```

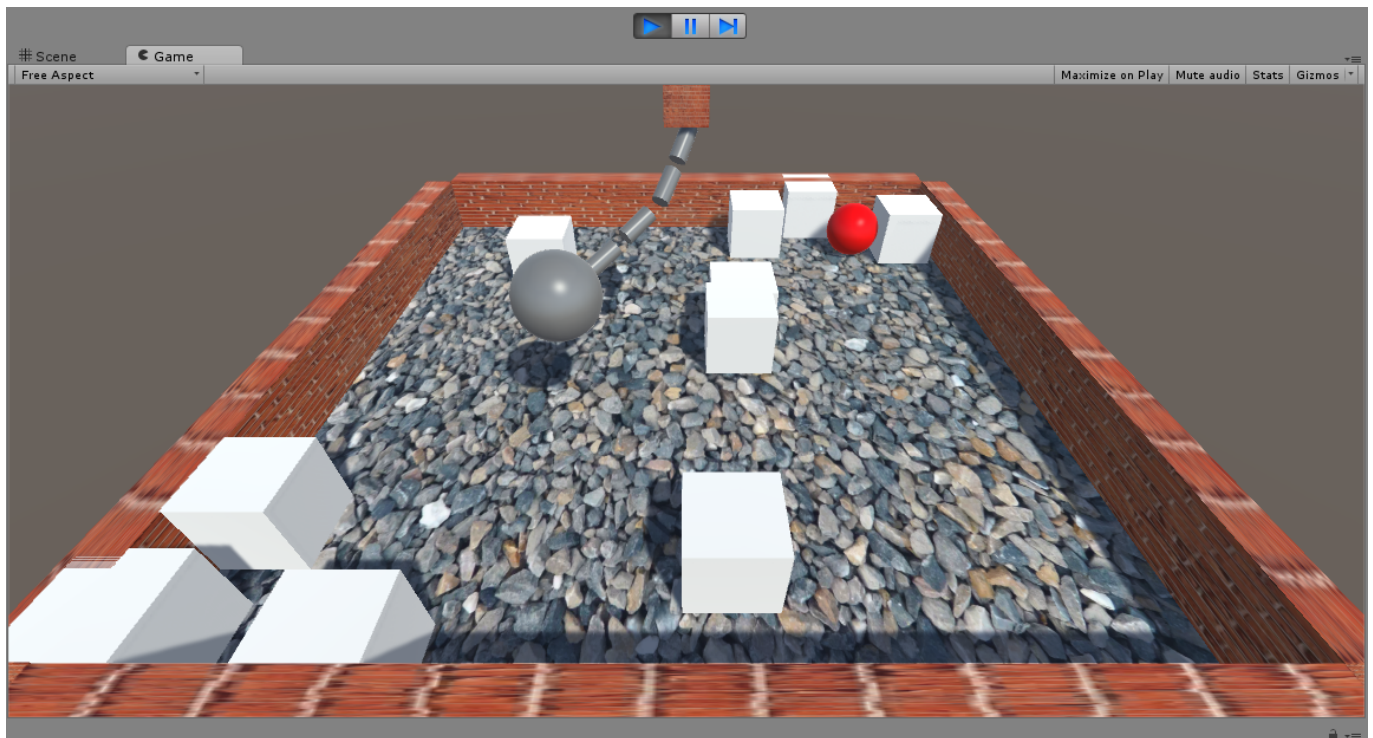
- Il est possible de générer une variable aléatoire comprise entre une valeur minimale *min* et une valeur maximale *max* grâce à la méthode *Range* de la classe *Random*.
- La fonction à utiliser pour créer un objet depuis un prefab est la fonction *Instantiate*. Ne vous occupez pas du troisième argument de cette fonction pour le moment, et utilisez *Quaternion.identity*).
- Dans le script CubeGenerator, nous utiliserons une variable *cubeCounter* qui contiendra le nombre de cubes présents dans la scène. A chaque création de cube, on incrémentera cette variable. Il faut également modifier le script *BallController* pour décrémenter cette valeur lorsque l'on détruit un cube.
- Prenez l'habitude d'utiliser la console pour debugger votre application. Il peut être judicieux d'écrire un message lorsque l'on détruit un cube, ainsi que lorsque l'on en crée un nouveau.

```
Debug.Log("Création d'un cube en (" + x + ", " + y + " , " + z + ")");
```



Mais... Ça marche !!!

Si tout va bien, on obtient quelque chose comme ça :



Des petits cubes générés aléatoirement

Pour aller plus loin

Il existe de nombreuses petites choses à améliorer dans cette application. Lorsque la boule de démolition percute un cube, elle peut rester coincée. Lorsque la boule que l'on déplace percute un cube, elle est ralentie (Mario ne ralentit pas lorsqu'il percute un champignon !!). Voici des pistes d'amélioration :

- Ne pas gérer la destruction des cubes par des collisions mais par des triggers.
- Déléguer cette gestion aux scripts des cubes, pas aux scripts de la boule ni de la boule de démolition.
- Ajouter un tag "Friend" à la boule de démolition (elle protège les cubes, elle est leur amie) et "Enemy" à la boule que l'on déplace (elle veut détruire les cubes, elle est méchante). Dans les scripts des cubes qui gèrent désormais les interactions, vous pouvez différencier le tag "Friend" qui ne va pas entraîner la destruction du cube, contrairement au tag "Enemy", qui lui détruira bien le cube.

