Date : 10/10/2023

Séance 6 – Moteur physique et gestion des collisions

Objectifs

- Introduire la gestion des collisions sur Unity
- Mise en pratique de concepts décrits sur le moteur de jeu Unity

Moteur de physique : le moteur de jeu vidéo Unity contient un ensemble des fonctionnalités pour nous aider à simuler la physique du monde réelle. Pour utiliser le moteur physique Unity, nous devons ajouter un composant « Rigidbody » à une instance d'un objet « GameObject ». Parmi les attributs plus utilisés d'un composant de type « Rigidbody », nous avons : « is kinematic », « use gravity » et « constraints ».

- **« Is Kinematic »** : cet attribut de type booléen définit si un objet « Rigidbody » est affecté par le moteur de physique.
- **« Use Gravity »** : cet attribut de type booléen définit si un objet « Rigidbody » est affecté par la gravité.
- **Les contraintes** sur un composant « Rigidbody » permettent de paramétrer les dégrées de liberté de son mouvement (ou déplacement). Les attributs « Freeze position » et « Freeze rotation » interdisent, respectivement, la translation et la rotation d'un objet.

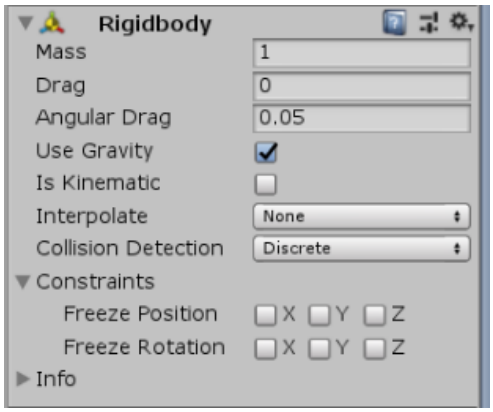

Figure 1. . Paramètres d'un composant « Rigidbody » sur la fenêtre « Inspector »

Gestion de collisions : le moteur de jeu Unity propose aussi des classes (ou des composants) pour gérer les collisions entre des objets. Pour avoir accès à cette fonctionnalité, une instance d'un GameObject doit contenir un composant de type « Collider ». Nous avons différents types de « Collider », par exemple, « SphereCollider » , « BoxCollider » et « MeshCollider ». Un composant « Collider » est comme une enveloppe (2D ou 3D selon le type de jeu) autour de l'objet. Elle peut être plus petite, exactement de la taille de l'objet ou plus grande. Nous devons la modéliser de façon à couvrir l'ensemble du volume autour de l'objet où nous voulons déclencher une collision. Pour modifier un « Collider », nous devons sélectionner l'objet « GameObject » cible, ensuite aller sur le menu « Inspector » et appuyer sur l'option « Edit Collider ».

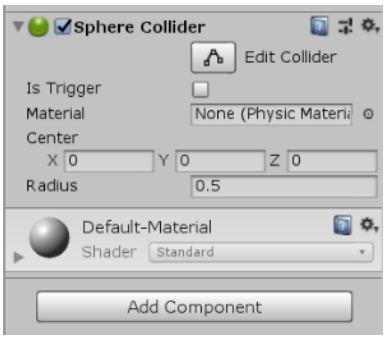

Figure 2. Paramètres d'un composant « SphereCollider » sur la fenêtre « Inspector »

Pour capturer des collisions entre des objets, nous utilisons des méthodes proposées par « MonoBehavior ». Les méthodes «Collider.OnCollisionEnter() », « Collider.OnCollisionStay()» et « Collider.OnCollisionExit() » sont appelées automatiquement lorsqu'un objet rentre en collision, reste en collision et fini sa collision avec un autre objet, respectivement. Ces méthodes possèdent comme argument un objet de type « collision » qui porte des informations relatives à la collision, telles qu'un pointer (non modifiable) vers l'objet « GameObject » tiers qui a fait une collision avec l'objet qui contient le script, les points de contact de la collision, entre autres informations. L'exemple ci-dessous montre comment capturer le nom de l'objet tiers qui vient de faire une collision avec l'objet cible. Cette méthode est appelée une seule fois pendant l'événement de la collision.

Exemple

```
private void OnCollisionEnter(Collision collision)
{
 Debug.Log("J'ai fait une collision avec l'objet:"
             + collision.rigidbody.name);
}
```

Les méthodes basées « *collision* » sont recommandées lorsque nous voulons capturer des collisions entre les objets affectés par le moteur physiques. Néanmoins, nous pouvons aussi capturer des collisions avec des objets qui ne sont pas affectés par le moteur physique.

Pour définir qu'un objet ne sera pas affecté par la physique lors d'une collision, nous devons cocher l'attribut « is trigger » de son composant « Collider ». Lorsque l'attribut « Is trigger » est activé, nous utiliserons les méthodes « OnTriggerEnter() », « OnTriggerExit() » et « OnTriggerStay() » pour traiter des collisions à la place des méthodes de la famille « OnCollision ».

Exemple

```
private void OnTriggerEnter(Collider other){
Debug.Log("J'ai fait une collision avec l'objet :" + other.name);
}
```

Méthode FixedUpdate : précédemment nous avons toujours utilisé la méthode « Update() » dans les scripts comportementaux pour capturer l'interaction avec l'utilisateur et faire des mises à jour de la scène, tel que le déplacement des objets. Néanmoins, dès que nous utilisons le moteur physique Unity, des objets « GameObject » avec un composant « Rigid Body », nous devons réaliser certaines actions dans la méthode « FixedUpdate » à la place de « Update ».

La méthode « Update » doit continuer à être utilisée pour des mises à jour ordinaires du jeu, comme le déplacement des objets non physiques, des chronomètres simples et du traitement des interactions avec l'utilisateur. La méthode « FixedUpdate » doit être utilisée pour toutes

les actions qui affectent des objets avec des composants « Rigidbody », car tous les calculs physiques sont faits par Unity juste après l'appel à cette méthode.

Notons que la méthode « Update() » est appelée une fois par « frame ». En revanche, la méthode « FixedUpdate() » peut être appelée zéro, une ou plusieurs fois par frame. Cette différence est due au fait que l'intervalle entre deux appels à la méthode « update » dépend du temps de traitement de la « frame » précédente. Par conséquent, les délais entre les appels à la méthode « Update() » seront fréquemment différents. En revanche, l'intervalle entre deux appels à la méthode « FixedUpdate » est paramétré et donc constant.

Le délai entre deux appels à la méthode « FixedUpdate » peut être modifié dans le menu « Édit > Settings > Time > Fixed Timestep ».

Le code source ci-dessous montre un exemple de répartition de code source entre les méthodes « Update » et « FixedUpdate » pour le déplacement d'un objet contrôlé par le joueur. L'implémentation proposée utilise la méthode « Input.GetAxis() » pour récupérer l'interaction avec l'utilisateur dans la méthode « Update ». L'information récupérée est stockée dans des attributs de la classe utilisée. Les valeurs de ces attributs seront utilisées dans les méthodes « deplacement() » et « rotation() » qui sont appelées dans la méthode « FixedUpdate() ». Les définitions de ces deux méthodes sont données plus tard dans la section « déplacement des objets physiques ».

```
private float deplacementAxeVertical ;
private float deplacementAxeHorizontal ;
private void Update(){
      deplacementAxeVertical   = Input.GetAxis ("Vertical");
      deplacementAxeHorizontal = Input.GetAxis ("Horizontal");
}
private void FixedUpdate(){
      deplacement() ;
      rotation() ;
}
```

Déplacement des objets physiques : lorsque nous utilisons des objets qui sont affectés par le moteur physique Unity, nous devons utiliser les méthodes proposées par la classe « Rigidbody » pour déplacer nos objets à la place de ceux définis par la classe « Transform ». Par exemple,

- *Rigidbody.MovePosition* (*Vector3*) : elle déplace un objet à partir du vecteur 3D passé en argument.

```
//implémentation de la méthode « deplacement() »
float vitesseDeplacement = 10f ;
Rigidbody objet_rb = GetComponent<Rigidbody>() ;
Vector3 mouvement = transform.forward * deplacementAxeVertical * vitesseDeplacement * Time.deltaTime;
objet_rb.MovePosition(objet_rb.position + mouvement);
```

- *Rigidbody.MoveRotation(Quaternion)* : cette méthode applique à l'objet la rotation passée en argument sous la forme d'un objet « Quaternion ». La classe « Quaternion » est utilisée pour représenter des rotations sur Unity.

La méthode « Rigidbody.MoveRotation(Quaternion) » a l'avantage de ne pas être affectée par le problème de « gimbal lock » (https://fr.wikipedia.org/wiki/Blocage_de_cardan).

```
// implémentation de la méthode « rotation » ;
float vitesseRotation = 5f ;
Rigidbody objet_rb = GetComponent<Rigidbody>() ;
objet_rb = GetComponent<Rigidbody>() ;
float rotation = deplacementAxeHorizontal * vitesseRotation * Time.deltaTime;
Quaternion q_rotation = Quaternion.Euler (0f, rotation, 0f);
objet_rb.MoveRotation (objet_rb.rotation * q_rotation);
```

Exercice 1 :

Téléchargez le projet « GestionCollision » sur la plateforme de cours et effectuez les actions proposées par les exercices 1 à 6.

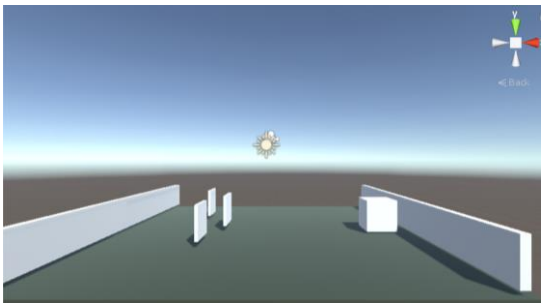

Figure 3. Scène proposée

1. L'objet « terrain » et l'un des murs tombent lorsque le jeu est lancé. Paramétrez-les de façon qu'il ne soit plus affecté par la gravité.
2. Le code source actuel déplace l'objet « joueur » à l'aide de son composant « Transform » de la méthode « Update ». Modifiez le code source pour utiliser les méthodes proposées pour le déplacement des objets physiques.

   a) Attention à bien répartir le code entre les méthodes « Update() » « FixedUpdate() », et préférez l'utilisation de la méthode « GetAxis() » à la place de la méthode « GetKey() ».

   b) Pensez aussi à modifier les contraintes du composant « Rigidbody » de l'objet joueur pour autoriser des changements de direction de déplacement.

3. Vous pouvez utiliser le code source ci-dessous pour arrêter le déplacement (attribut « velocity ») et la rotation (attribut « angularVelocity) du composant Rigidbody e l'objet "joueur" après son déplacement.

```
« rigid_body_objet ».velocity            = Vector3.zero;
« rigid_body_objet ».angularVelocity     = Vector3.zero;
```

4. Créez des obstacles pour l'objet « joueur » comme la montre la Figure 3. Faites en sorte que lorsque l'objet « joueur » rentre en collision avec l'un des obstacles, cet obstacle soit affecté par le moteur de physique et qu'il change de couleur selon les règles suivantes :
   - Jaune : moment de la collision et collision en cours
   - Couleur originale : collision finie

Conseil : commencez par afficher un message sur la console lorsqu'une collision est détectée pour une méthode donnée de traitement de collision. Ensuite, essayez de capturer les différentes phases de collision. Une fois que vous avez terminé les étapes précédentes, passez au changement de couleurs.

5. Entourez la scène des murs et ajoutez plusieurs objets de type « obstacle ». Paramétrez les contraintes des composants « Rigidbody » afin d'éviter que les obstacles et les murs tombent de la scène.

Perfectionnement

6. Créez un nouvel objet basé sur le « GameObject » de type « Capsule ». Lorsque l'objet « joueur » traverse ce nouvel objet, l'objet « joueur » devient plus grand et l'objet traversé disparaît. L'objet traversé ne doit pas subir les effets du moteur de physique.
7. Finalement, lorsque l'objet « joueur » a consommé un objet « Capsule », il pourra détruire un objet « obstacle ». Le jeu terminera dès qu'il n'y a plus d'obstacle dans la scène. Attention, lors d'une collision, l'objet « Collision » passé en argument contient une instance de GameObject. En revanche, cette instance est en mode lecture seulement.

N'hésitez pas à utiliser la méthode « Find » de la class « GameObject » pour chercher l'instance à détruire.

Exercice 2 :

Reprenons maintenant notre projet Unity de la séance précédente (les animaux de la ferme). Jusqu'à présent, vous avez placé manuellement les animaux dans la scène avant le démarrage du jeu. Dans cet exercice, votre premier objectif sera de gérer les collisions entre les animaux et la nourriture lancée par le joueur. Ensuite, nous allons améliorer notre jeu afin de le munir de la capacité de créer des animaux dans la scène de façon automatique, afin de simuler un effet des vagues « d'ennemies ». À la fin de ce TD, la position initiale de chaque animal sera choisie aléatoirement, ainsi que le modèle utilisé pour le créer et les animaux seront créés dans la scène automatiquement et à un intervalle régulier.

Modèles d'animal :

1. Créez un script en utilisent les méthodes de détection de collision vues pendant la séance afin de faire disparaître de la scène les animaux lorsqu'ils sont nourris par le personnage (collision entre un animal et la nourriture) .
2. Ajoutez votre script ainsi qu'un composant « Collider » au modèle (« prefab ») de chaque animal utilisé dans votre scène. Vous devez ajouter aussi un objet de type « Collider » au modèle de projectile.

   - Le type d'objet « Collider » choisi doit être adapté à l'objet utilisé.
   - Vous devez créer un nouveau modèle « prefab » pour chacun de trois animaux choisis contenant le script créé, ou mettre à jour vos modèles précédents.

Gestion de la création des animaux (Animal Spawn Manager)

3. Créez un objet vide « Empty Game Object », ainsi qu'un script nommé « Gestionnaire Jeu » pour gérer la création des animaux. Associez ce script à l'objet créé.
4. Rajoutez une méthode, nommée « Creation Animal », au script « Gestionnaire Jeu » avec l'objectif de créer un objet à partir des arguments suivants : un modèle « prefab », une position et une rotation dans la scène.

   - Vous pouvez utiliser la méthode « Instantiate » pour créer les objets de type "GameObject".
   - Le code source attendu est similaire à celui utilisé pour créer des projectiles.
   - Le modèle « prefab » à utiliser doit être un attribut de votre script.

5. Utilisez la méthode « Creation Animal » pour créer une instance d'un modèle "prefab" d'un animal choisie par vous, à chaque fois que nous appuyons sur la touche « C ». La position de l'animal doit être choisie au hasard.

Placement au hasard des animaux dans la scène : vous pouvez fixer deux dimensions de la position choisie pour placer les animaux dans la scène (e.g., Y et Z), et tirer au hasard la dimension restante en utilisant la méthode « Random.Range(…) ». Dans le cas de notre jeu, vous pouvez tirer au hasard la dimension X du vecteur qui contient la position de l'objet.

Je vous conseille de définir l'intervalle de valeurs fournies à la fonction « Range » en utilisant des attributs de votre script.

6. Lorsque vous avez validé le fonctionnement de votre script « Gestionnaire Jeu» pour la création des animaux à la demande, modifiez votre code source afin d'utiliser la méthode « InvokeRepeating » pour créer les animaux dans la scène de façon totalement automatique à un intervalle de temps régulier.

La méthode « InvokeRepeating » permet de faire des appels réguliers à une fonction. Cependant, la fonction appelée ne peut pas atteindre des arguments.

https://docs.unity3d.com/ScriptReference/MonoBehaviour.InvokeRepeating.html

Jusqu'à présent, nous avons utilisé toujours le même modèle d'animal, et nous avons créé plusieurs instances de ce modèle dans la scène, placés de façon aléatoire.

7. Modifiez votre code source afin de choisir le modèle d'animal à utiliser pour créer un objet aléatoirement, parmi un ensemble de modèles indiqués au préalable (fenêtre « Inspector »).

- Vous pouvez modifier l'attribut de type « GameObject » qui indiquait un modèle « prefab » à utiliser afin qu'il puisse comporter une liste d'objets de type « GameObject ».
- N'oubliez pas d'indiquer les modèles « prefab » à utiliser par votre script.

Considérations finales

Une partie des exercices de ce TD a été adaptée de l'unité 2 du Parcours Unity Learn - Junior Programmer. N'hésitez pas à visiter leur formation en ligne afin d'avoir accès au cours complet, ainsi qu'à avancer en autonomie sur les sujets que n'ont pas été traité en cours.

Références

- https://docs.unity3d.com/ScriptReference/Rigidbody-isKinematic.html
- https://docs.unity3d.com/ScriptReference/Rigidbody-constraints.html
- https://docs.unity3d.com/ScriptReference/Collision.html
  https://docs.unity3d.com/ScriptReference/Rigidbody.MovePosition.html
- https://docs.unity3d.com/ScriptReference/Quaternion.html
- https://docs.unity3d.com/ScriptReference/Random.Range.html