Jeux et simulations

Éric Wenaas

420-4GP-BB

Module 4 – Mouvement première personne et singleton

Version 2.2

Le mouvement première personne

Paramètres locaux vs. paramètres globaux

Les propriétés du Transform contiennent deux types de valeurs pour les objets de jeu. Les valeurs locales et les valeurs globales. La valeur globale est relative au monde. La valeur locale est par rapport au parent. Il faut donc faire attention lorsqu'on déplace un objet qui a un objet parent. Un objet va suivre son parent. Si le parent bouge, l'objet va également bouger. Le fait de déplacer un objet à partir de son parent modifiera ses valeurs globales mais non ses valeurs locales. La solution que nous allons implémenter pour le mouvement première personne permettra de saisir la distinction entre les deux types de valeurs du Transform.

Principes du mouvement première personne

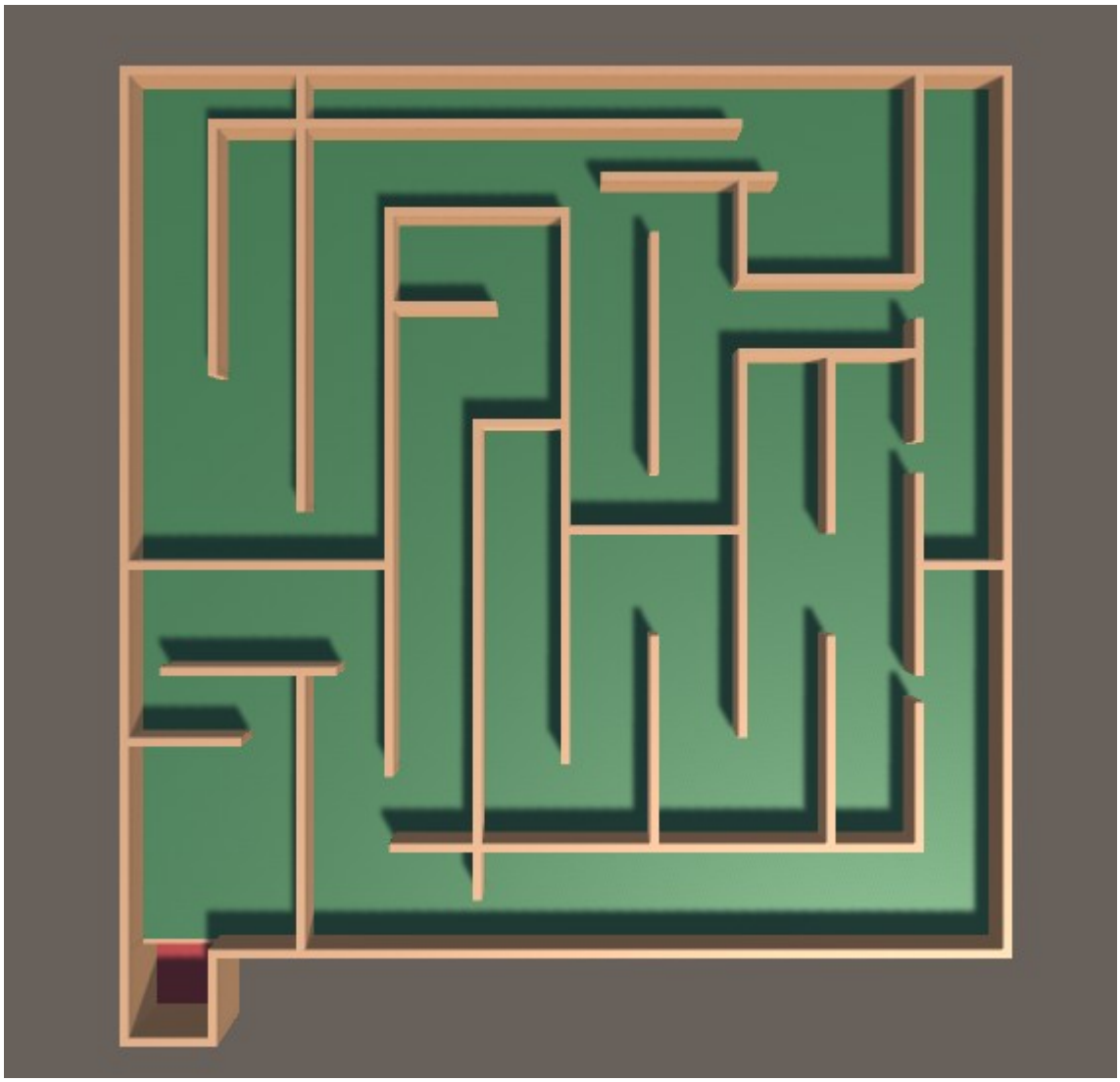
Le type de mouvement première personne est très populaire dans le domaine du jeu. Ce type de mouvement nous donne l'impression d'être un personnage dans le jeu. Les principes de base pour réaliser ce type de mouvement sont :

- La caméra est attachée à l'objet de jeu principal (le joueur). La caméra est donc un enfant de l'objet joueur. Ce qui fait que la caméra va suivre le joueur. Normalement, le joueur sera très peu visible dans ce type de mouvement.
- La souris permet de regarder dans diverses directions. Pour regarder dans différentes direction, on fait des rotations soit sur le joueur, soit sur la caméra
- Le clavier est utilisé dans le but de se déplacer, normalement avec les touches WASD.

Par la suite, il est possible d'y ajouter certaines options, tel que la possibilité de courir ou encore de sauter. Selon les paramètres du jeu, on peut ensuite ajouter des projectiles ou encore une certaine interaction avec les objets du jeu. Cette série d'exercice nous permettra de mettre en pratique certains de ces éléments.

Exercice #1

Reprenez le labyrinthe que nous avions fait dans le module 1. Ce labyrinthe vous est donné sous forme de package Unity. Nous allons faire un déplacement première personne.



Création du joueur

- Créez un Prefab pour le joueur. Prenez une capsule 3D comme objet principal.
- Placez également une caméra à l'intérieur de cet objet

Verrouillage de la souris

Créez un objet vide et nommez-le GameManager. Idéalement, la souris devrait être verrouillée pour avoir un déplacement fluide. Créez ce script et attachez-le à votre contrôleur de jeu :

```
public class CursorLock : MonoBehaviour
{
    Message Unity | 0 références
    void Start()
    {
        Cursor.lockState = CursorLockMode.Locked;
    }

    Message Unity | 0 références
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape)) {
            Cursor.lockState = CursorLockMode.None;
        }
    }
}
```

Rotation sur la caméra et sur le joueur

Nous allons réaliser la rotation de la façon suivante. Les rotations de gauche à droite, sur l'axe Y seront faite sur le joueur. Puisque la caméra est un enfant du joueur, elle suivra le joueur. Les rotations de haut en bas seront faites sur la caméra car on ne veut pas que le joueur tourne sur cet axe. Les rotations sur le joueur seront des rotations globales dans le monde alors que les rotations sur la caméra seront des rotations locales, en fonction de la position du joueur.

- Faire un script VueSouris.cs et l'attacher à la caméra. Pour accéder à l'objet vide, on peut utiliser transform.parent.gameObject ou encore se déclarer une variable sérialisée[1].
- Faire tourner le joueur et la caméra en fonction du déplacement de la souris. (Il faut utiliser Input.GetAxis) Les paramètres permettant de voir le déplacement de la souris se retrouve dans **Edit → Project Settings → Input**
- Faire les rotations en fonction de ce qui a été lu.
- La rotation sur la caméra devrait être contrainte entre deux bornes avec la fonction Mathf.Clamp.

Attention : Le joueur tourne autour de l'axe Y et la caméra autour de l'axe X[2].

1 Avec [SerializeField].

2 À moins que vous désiriez qu'il y ait une rotation sur cet axe.

Character Controller

Le composant Character Controller est particulièrement utile pour contrôler le mouvement d'un personnage dans un style de mouvement première personne. Sa principale caractéristique est que le personnage n'aura pas de Rigidbody non kinématique. Dans l'éventualité où on désire que des lois physiques soient applicables au joueur, il ne faudrait pas utiliser le CharacterController.

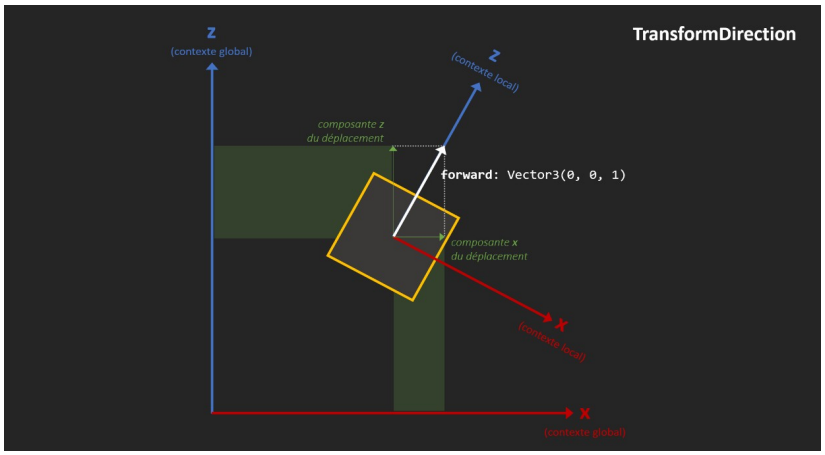
Le personnage ne pourra néanmoins pas entrer dans les murs et les collisions seront détectées.

Quand on utilise un CharacterController, il faut déplacer l'objet strictement avec les méthodes du CharacterController. Le CharacterController devrait donc être désactivé si on veut déplacer un joueur autrement que par le CharacterController.

Exercice #2

Vous devez ajouter un CharacterController au joueur. En lisant avec Input.GetAxis() les valeurs pour les axes *Horizontal* et *Vertical,* faites en sorte de déplacer votre joueur en utilisant le Character Controller.

Remarquez que le joueur ne se déplace pas toujours dans la direction attendue. La raison est que les valeurs des axes sont calculés en fonction du positionnement global. Il faut adapter ces valeurs en fonction de la direction dans laquelle le joueur se dirige (son vecteur forward).



Dans un deuxième temps, consultez la méthode Transform.TransformDirection() pour que les axes soient adaptés à la rotation du joueur.

Faites également en sorte que le joueur aille plus vite lorsqu'on appuie sur la touche majuscule de gauche. Il suffit dans ce cas d'augmenter la vitesse.

Exercice #3

Le CharacterController a une propriété isGrounded. Cette propriété permet de savoir si l'objet de jeu est sur le sol. Nous allons utiliser cette propriété pour implémenter une fonction de saut. On va également l'utiliser pour permettre au joueur de tomber s'il n'est pas sur le sol.

Pour sauter, on fait un mouvement vers le haut, sur l'axe y. Pour tomber, on déplace sur le même axe mais vers le bas.

Si le joueur n'est pas sur le sol. Il ne devrait pas pouvoir ni sauter, ni se déplacer.

Exercice #4

Dans la zone d'arrivée, nous allons replacer le personnage à sa position initiale. Pour la zone d'arrivée, il faudra modifier votre scène afin que le déplacement se fasse maintenant avec le CharacterController. Pour la zone d'arrivée, il faudra utiliser la méthode OnControllerColliderHit().

De plus, pour remettre le joueur dans sa position initiale, vous devrez désactiver le CharacterController (avec la propriété *enabled*)

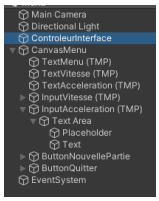
Le patron singleton

Dans cette section nous verrons en quoi consiste le patron singleton. Le contexte d'application de ce patron sera le suivant :

Vous devez créer, dans une autre scène, le menu suivant :

Pour créer le menu, vous devrez utiliser les éléments d'interface UI de Unity. Ces éléments sont relativement simples à placer et à configurer. Il y a quelques façons d'implémenter une action sur un élément d'interface. Une des plus simple est de rattacher un événement aux différents éléments en question.

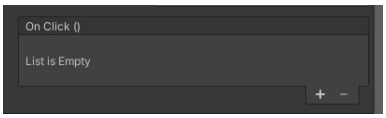
Voyons pas à pas comment implémenter le bouton Quitter :

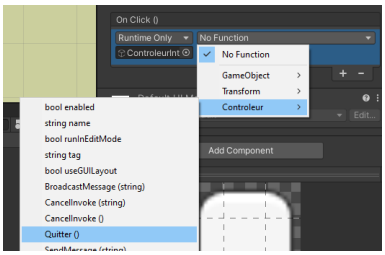
1. Créez un objet vide se nommant ControleurInterface : 

2. Attachez-lui un Scripts se nommant Controleur qui contiendra cette méthode :

```
public void Quitter()
{
#if UNITY_EDITOR
    EditorApplication.isPlaying = false;
#else
    Application.Quit();
#endif
}
```

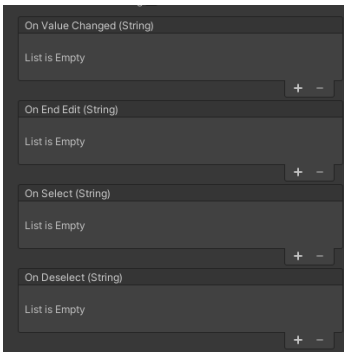
3. Attachez l'événement OnClick() du bouton Quitter à la méthode Quitter() de l'objet de jeu ControleurInterface :





Ce qui aura pour effet d'appeler la méthode Quitter() quand on clique sur le bouton.

Chaque élément d'interface a ses propres événements qu'il peut gérer. Par exemple, pour le InputField, vous avez les événements suivants :



Ce menu permet à l'utilisateur de spécifier la vitesse et le facteur d'accélération dans le jeu du labyrinthe. Par défaut, ces valeurs seront de 15 et 1.5. Lorsque l'utilisateur clique sur le bouton Nouvelle partie, on doit charger la scène du labyrinthe. Quand il clique sur Quitter, on quitte simplement le logiciel.

Dans le cadre du jeu du labyrinthe, si l'utilisateur fait la touche d'échappement, on charge la scène du menu.

Exercice #5

Réalisez le menu ainsi que la navigation entre le jeu et le menu. Le traitement pour les valeurs sera fait dans l'exercice suivant.

Singleton

La difficulté est de faire en sorte que les valeurs indiquées par l'utilisateur seront utilisées dans le labyrinthe. Le problème est que les valeurs doivent survivre au chargement de la nouvelle scène. Pour ce faire, nous allons utiliser le patron singleton.

Le singleton est un patron de conception permettant de résoudre le problème demandant de s'assurer qu'il n'y ait dans un système qu'une seule instance d'une classe. Afin de s'assurer que ce sera le cas, nous respecterons les règles suivantes :

- Le constructeur est privé
- La classe contient une instance statique et privée de la classe
- La classe contient une méthode statique permettant d'obtenir une référence vers l'instance unique.
- Les autres méthodes de la classe ne seront pas statiques mais s'appliqueront simplement sur l'instance unique.

Voici le code de base d'un Singleton :

```
public class Singleton
{
  private static Singleton _instance = new Singleton();
  private Singleton()
  {
    // Initialiser les attributs du singleton
  }

  public static Singleton Instance
  {
    get {return _instance;}
  }
  // Ajouter les propriétés du Singleton ainsi que différentes méthodes
  // pertinentes au contexte
}
```

Naturellement dans ce code, le nom de la classe doit être remplacé par un nom représentatif.

Critiques sur le singleton

Le singleton est un patron qui peut être fort utile s'il est utilisé judicieusement. Toutefois, si on y pense bien il s'agit simplement d'une variable globale. C'est un objet qui est accessible à partir de toutes les classes du système et qui peut être modifié par toutes les classes du système. On pourrait théoriquement remplacer un singleton par une variable statique et publique dans une classe, ce qui donnerait essentiellement la même chose.

Le singleton est donc une solution dont il ne faut pas abuser.

De plus, dans son implantation de base, le singleton n'est pas *thread-safe*. Il faudrait donc le munir d'un verrou dans un logiciel utilisant massivement les *threads*. Ceci implique des coûts au niveau de la performance.

Dans l'éventualité où la valeur du singleton va être consultée à plusieurs endroits mais modifiée à un seul et qu'elle a des impacts majeurs sur le comportement du système, un singleton devient pertinent. Ça nous évite de passer un paramètre partout. **Dans le cas ou on utilise le singleton dans le but de permettre à tous les objets d'accéder à n'importe quoi n'importe quand, le singleton ne devrait pas être utilisé. Il serait alors utilisé comme une grosse variable globale.**

Exercice #6

Faites en sorte que les valeurs spécifiées par l'utilisateur pour la vitesse et le facteur d'accélération soient conservées dans un singleton modifié à partir du menu et consulté dans la scène de jeu.

Exercice #7

Dans le contexte de Unity, il est possible qu'un singleton ait besoin d'être rattaché à un objet de jeu. Dans ce cas, la création de l'objet créera une instance de la classe du singleton. Si une scène est rechargée plusieurs fois, plusieurs objets seront créés. Ce qui va à l'encontre du principe du patron singleton. Dans ce contexte, il faudra adapter notre patron à Unity.

La stratégie que nous allons prendre est la suivante :

1. Dans la propriété Instance, on cherche l'objet avec FindObjectOfType
2. S'il n'est pas trouvé, on crée un nouveau GameObject et on lui ajoute un composant du type du Singleton.
3. On implémente la méthode Awake() avec ce code :

```
if (_instance == null)
{
    _instance = this;
    DontDestroyOnLoad(gameObject);
}
else
{
    Destroy(this);
}
```

Renommez le singleton GameManager et faites en sorte qu'il soit rattaché au contrôleur d'interface dans le menu et à l'objet GameManager dans le labyrinthe. Une seule instance de ce composant doit exister. Si on découvre qu'il a déjà été créé, on détruit simplement l'objet.