ATI L3 2021-22

Support de cours Unity

Cours 2 : Conditions, Inputs, Vector3 (approfondi)

14/10/19

Présentation

Dans ce cours, nous allons voir trois points fondamentaux qui vont pour servir, aussi bien pour le TD que pour vos futurs projets. Je vous parlerai d'abord des conditions **if**, que vous reconnaîtrez du cours de la semaine dernière. J'adresserai ensuite la manière de laquelle il faut gérer les **Inputs** dans Unity, à savoir reconnaître les saisies du joueur sur le clavier, la souris, la manette… Je terminerai ce pdf avec quelques opération de **Vector3** un peu plus complexes que ce que nous avons vu la dernière fois, et quelques autres opérations mathématiques possibles.

Conditions

Les conditions servent à n'exécuter certaines opérations que dans certaines circonstances et non à chaque fois que la fonction Update sera appelée. Il en existe plusieurs types, mais au final, elles vérifient toutes la même chose : si quelque chose est vrai.

Condition « Si X Alors Y »

Ici nous vérifions une condition simple avec un Bool.

(rappel : un Bool ne peut avoir qu'une valeur égale à **vrai** ou **faux**)

```
public bool maValeur = true;

void Start() {
    if(maValeur == true)
    {
        Debug.Log("maValeur est vrai");
    }
}
```

Notez bien le double signe égal (==) entre les parenthèses du if. Il est essentiel au fonctionnement d'une condition : s'il n'y avait qu'un seul signe égal, on ne vérifierai pas la valeur, mais on lui assignerai la valeur pour laquelle on essaye de le vérifier !

Notez surtout que dans les exemples ici je ne montre que des conditions avec des Bool, mais la condition peut aussi se faire avec une valeur :

```
int i = 5;
if (i < 10) {
    // Faire Des Trucs
}
```

La structure d'une condition sera donc toujours la même :

if (ma condition entre parenthèses)
{

ce que je dois faire si la condition est vérifiée, entre accolades ;
}

Condition « Si X Alors Y Sinon Z »

A l'aide du mot-clé « else », nous pouvons faire une alternative à la condition. C'est à dire ce qui est à faire spécifiquement dans le cas où la condition n'est pas vérifiée.

```
public bool maValeur = true;

void Start() {
    if(maValeur == true)
    {
        Debug.Log("maValeur est vrai");
    }
    else
    {
        Debug.Log("maValeur est fausse");
    }
}
```

Condition « Si X ou Y Alors Z »

Le double symbole barre verticale « || » (AltGr + 6 sur un clavier Azerty classique) permet de demander une espèce de « ou bien ».

```
public bool maValeur = true;
public bool monAutreValeur = true;

void Start() {
    if(maValeur == true || monAutreValeur == true)
    {
        Debug.Log("maValeur est vrai");
    }
}
```

Dans cet exemple si, la condition est vérifiée même si un seul de mes Bools est vrai ! Pratique non ?

Condition « Si X et Y alors Z »

Le double symbole esperluette « && » permet de vérifier en une seule ligne que deux conditions sont vraies.

```
public bool maValeur = true;
public bool monAutreValeur = true;

void Start() {
    if(maValeur == true && monAutreValeur == true)
    {
        Debug.Log("maValeur est vrai");
    }
}
```

Conditions numériques

Jusqu'ici mes seuls exemples étaient avec des Bools, mais on peut aussi réaliser une condition pour vérifier si une valeur numérique est supérieure, inférieure, égale, …, à une autre.

```
public float maValeur = 5f;

void Start() {
    if(maValeur < 10f)
    {
        Debug.Log("ma condition est vrai");
    }
}
```

Notez qu'ici le symbole « < » ne sert qu'à vérifier si la valeur est strictement inférieure à 10. Le symbole « <= » peut être utilisé pour vérifier « inférieur ou égal à ».

De même, le symbole « > » sert à vérifier si une valeur est strictement suppérieure, et « >= » peut être utilisé pour un « suppérieur ou égal à ».

```
public float maValeur = 5f;

void Start() {
    if(maValeur > 0f)
    {
        Debug.Log("ma condition est vrai");
    }
}
```

/! Nous aurons également le « == » que nous avons vu avec les Bool plus tôt, qui permet de vérifier si une valeur est strictement égale à une autre. Il existe aussi la combinaison « != » qui vérifie si une valeur est strictement *différente* de celle qu'on prend en référence.

Inputs

Input d'une touche

Les Inputs servent à récupérer les saisies du joueur. Une détection d'inputs se fait habituellement à l'aide d'une condition, de la façon suivante :

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

    if (Input.GetKey (KeyCode.A)) {
        Debug.Log ("J'appuie sur la touche A !");
    }

}
```

La condition est donc composée de deux éléments :

- **Input.GetKey** : Détermine quel type d'input nous utilisons. Le GetKey vérfie si la touche est pressée au moment où la condition a lieu. Dans le cas ci-dessus, ça veut dire que je vais avoir le print à *chaque frame* pendant que la touche A est appuyée.
    - Si on veut détecter *l'instant* où la touche est appuyée, il faut effectuer un Input.GetKeyDown(touche).
    - De même, on peut détecter l'instant où la touche est relachée avec un Input.GetKeyUp(touche).

```
void Update () {

    if (Input.GetKeyDown (KeyCode.A)) {
        Debug.Log ("J'ai appuyé sur A !");
    }
}
```

- **KeyCode.A**:la touche que l'on souhaite détecter.
    - On peut y rentrer la touche que l'on veut : KeyCode.UpArrow, KeyCode.Space, KeyCode.LeftShift, KeyCode.Keypad4, KeyCode.Mouse0, etc.

```
void Update () {

    if (Input.GetKeyDown (KeyCode.F5)) {
        Debug.Log ("J'ai appuyé sur F5 !");
    }
}
```

    - Il est également possible de se créer une variable de type KeyCode en public afin de la modifier dans la fenêtre Inspector.

```
public KeyCode maTouche;

void Update () {

    if (Input.GetKeyDown (maTouche)) {
        Debug.Log ("J'ai appuyé sur ma touche !");
    }
}
```

        - Ca peut être utile pour utiliser un même script pour deux joueurs, et leur donner chacun une touche différente d'interaction.

Input d'un axe

On va avoir un petit souci en revanche. Si on décide de faire un script de déplacement de personnage (ou, à tout hasard, d'un vaisseau spatial), on pourrait finir avec quelque chose comme ça :

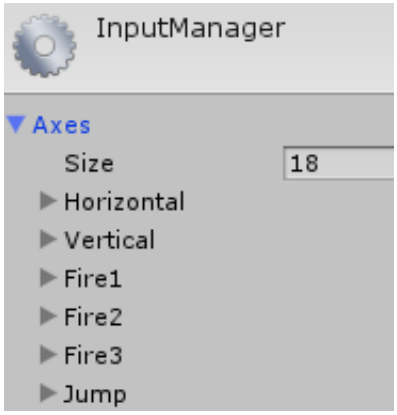
```
void Update() {
    if(Input.GetKey(KeyCode.LeftArrow)) {
        transform.Translate(Vector3.left);
    }
    if(Input.GetKey(KeyCode.RightArrow)) {
        transform.Translate(Vector3.right);
    }
    if(Input.GetKey(KeyCode.UpArrow)) {
        transform.Translate(Vector3.forward);
    }
    if(Input.GetKey(KeyCode.DownArrow)) {
        transform.Translate(Vector3.back);
    }
}
```

Mais c'est long et on a l'impression de recopier plusieurs fois la même chose.

Unity nous permet une alternative. Elle porte le doux nom fleuri et mélodieux de : **GetAxis.** Le GetAxis nous permet d'utiliser les Inputs en tant que float et non en tant que bool. Ca peut ne pas être évident au premier abord car la méthode est différente.

Un GetAxis ne récupère pas les informations d'une seule touche comme le GetKey. Un Axis est défini selon deux touches, qui vont être les valeurs positives et négatives de cet axe. Par exemple, l'axe « Horizontal » a pour touche positive (+1) la flèche de droite et pour touche négative (-1) la flèche de gauche.

Les Axis peuvent être configurés dans le Input Manager de Unity. Vous le trouverez dans **Edit > Project Setting > Input**. Vous y trouverez entre autres les axes « Horizontal » et « Vertical ».



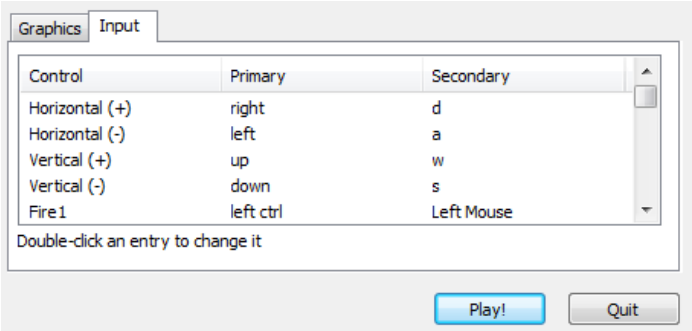
Notez que les Axis permettent d'avoir deux touches possibles comme touche positive ou négative de l'axe, et que les touches alternatives de « Horizontal » et « Vertical » dont casés sur les claviers anglophones : c'est-à-dire les touches WASD. Si vous voulez vous déplacer en ZQSD, il va falloir changer ces touches alternatives.

L'utilisation d'un GetAxis se fait de la sorte :

```
void Update() {
    float h = Input.GetAxis("Horizontal");
    float v = Input.GetAxis("Vertical");
    // Les axes nous retournent des float. On va donc les stocker dans des float pour ensuite
    // pouvoir les utiliser dans un Vector3 qu'on va utiliser en Translate.
    Vector3 mouvement = new Vector3(h, 0f, v);
    transform.Translate(mouvement * Time.deltaTime); // il faut ajouter un deltaTime pour plus de swag
}
```

Les Axis ont d'autres intérêts :

- Comme on les appelle par leur nom, on peut les utiliser d'importe quand et gérer leurs touches d'Input séparément dans le Input Manager.
- Les Axis sont modifiables dans **l'executable** d'un jeu Unity ! Donc le joueur peut modifier les touches assignées à chaque action du jeu si le layout ne lui convient pas !



Le GetAxis renvoie toutes les subtilités d'input entre -1 et 1 (notamment pour l'inclinaison des joysticks). Mais des fois on ne veut pas avoir à s'embêter avec ça, et on peut utiliser le GetAxisRaw, qui lui renvoie uniquement -1, 0, ou 1.

Autres opérations : maths et Vector3

Je vais vous parler ici de quelques autres choses qu'on peut facilement faire dans Unity : incrémentation d'une valeur, calcul d'une valeur aléatoire, LookAt et MoveTowards, Lerp, et calcul de distances.

Incrémentation d'une valeur

Je vous en ai un peu parlé au dernier cours : une variable peut être utilisée dans un calcul pour sa propre valeur :

```
private int banane = 0 ;

void Update() {

        banane = banane + 7 ;

}
```

Ceci va augmenter de 7 la valeur de ma variable banane à chaque Update( ).

Mais la même chose peut être écrite comme ceci :

```
private int banane = 0;

void Update() {
    banane += 7;
}
```

Et de même, nous avons :

```
private int banane = 0;

void Update() {
    banane -= 1;
    banane *= 5;
    banane++;
}
```

« -= » décrémente la variable, et « *= » multiplie la variable. « ++ » rajoute 1 à la valeur de la variable.

Calcul d'une valeur aléatoire

Il existe dans Unity une fonction Random qui permet de calculer des nombres aléatoires.

Tout d'abord, il existe le Random.value :

```
float a = Random.value;
```

Ceci donne une valeur aléatoire entre 0,0 et 1,0. En la multipliant par 100 vous aurez des pourcentages aléatoires, par exemple.

Mais il est aussi possible de directement calculer des nombres aléatoires entre deux valeurs. On va utiliser pour ça le Random.Range (valeur min, valeur max).

Mais **ATTENTION**, il existe deux versions de ce Random.Range, et la différence entre les deux est très subtile ! En effet, le Random.Range peut servir à calculer une valeur aléatoire entière (un int) ou décimale (un float). Ce qui va déterminer cette distinction est la nature des valeurs comprises entre les parenthèses du Range.

- Si les valeurs entre les parenthèses sont entières (des int), la valeur calculée sera également un int. A noter qu'avec les entiers, la valeur max est **excluse**. Donc qu'on choisisse de la stoquer dans un int ou un float, l'exemple ci dessous :

```
int randInt = Random.Range (-5, 5);
float randFloat = Random.Range (-5, 5);
```

  ne peut au final retourner comme valeur que -5, -4, -3, -2, -1, 0, 1, 2, 3, ou 4.

- Si les valeurs entre les parenthèses sont décimales (des float), la valeur calculée sera décimale, et comprise entre les deux valeurs, la maximale étant **incluse**. L'exemple suivant :

```
float randFloat2 = Random.Range (-5f, 5f);
```

  rendra un nombre aléatoire compris entre les deux valeurs, -4.3, 2.1, …

Dans le TD, je vous demande de calculer une position aléatoire. Vous avez deux possibilités pour ça :

```
Vector3 pos = Vector3.zero;
pos.x = Random.Range (-5f, 5f);
```

ou :

```
float r = Random.Range (-5f, 5f);
Vector3 pos = new Vector3 (r, 0, 0);
```

et ensuite seulement faire un « transform.position = pos ; ».

Attention ! Vous ne pouvez <u>en aucun cas</u> faire un « transform.position.x = Random.Range(-5f, 5f) ; » !

Ca ne fonctionnera pas car la position du Transform n'est pas conservée dans votre script. On peut la récupérer, mais pas la modifier directement. Si on veut modifier la position il faut tout changer d'un coup avec un Vector3. Pourquoi ? Pour éviter qu'on puisse tout modifier n'importe comment et qu'on fasse des bêtises.

LookAt

Le LookAt permet à un objet de se tourner vers une position donnée. L'orientation « avant » de l'objet sera son axe Z positif local.

```
public Transform target;

void Update() {
    transform.LookAt(target.position);
}
```

MoveTowards

Le MoveTowards permet de réaliser un mouvement qui va d'un point A à un point B, de façon constante, avec une valeur vitesse.

```
public Transform target;

void Update() {
    transform.position = Vector3.MoveTowards(transform.position, target.position, Time.deltaTime);
}
```

Dans cet exemple, le point A est la position actuelle de l'objet, ça revient à un déplacement constant vers le point B. La valeur de vitesse dépend du deltaTime.

Lerp

Lerp signifie « **L**inear Int**erp**olation ». Pas évident comme acronyme. Il permet donc de faire une interpolation linéaire d'un point vers un autre. Il est très similaire au MoveTowards, mais permet d'avoir une progression plus douce au début, accélérée au milieu, et à nouveau douce à la fin.

Beaucoup plus doux, il va souvent être utilisé pour les mouvements de caméras.

```
public Transform target;

void Update() {
    transform.position = Vector3.Lerp(transform.position, target.position, Time.deltaTime);
}
```

Distance entre deux points A et B

Bien que ce soit un opérateur de Vector3, la distance n'est qu'une seule valeur, et se stocke par conséquant dans un float !

```
public Transform A;
public Transform B;
float distance;

void Update() {
    distance = Vector3.Distance(A.position, B.position);
}
```