TD5: Communication entre script, GUI et scènes

Introduction

L'objectif de ce TD est d'apprendre les bases d'une application "complète". Pour cela, nous allons créer un mini-jeu au gameplay rudimentaire ainsi qu'un menu d'ouverture.
Le but du jeu est de lancer une boule sur un ensemble de quille de façon à maximiser le nombre de choc **entre elles**.
Nous aborderons notamment :

- les mécanismes de communication entre objet
- le GUIScript pour créer des interfaces utilisateurs
- le système de tag de Unity
- le passage entre les scènes

Mise en place de la scène

Importer le package fourni avec le TD. Ce package contient une scène dans laquelle sont placés un plan "Sol", une caméra qui vous permettra de vous déplacer sommairement dans l'environnement et une boule de bowling "Ball".

Nous allons maintenant ajouter des actions sur la boule de bowling qui seront accessible via des boutons sur la sphère elle-même.

- Une action "Grasp" pour attraper la boule
- Une action "Release" pour lâcher la boule.
- Une action Throw pour la jeter.

Créez un scrip BallBehavior que vous ajouterez à la boule de bowling.
Le script ici a uniquement vocation à implémenter les comportements de ces actions. Dans un souci de modularité, nous créérons le menu dans un script indépendant.
Nous ajoutons un boolean *bAvailable* qui passe à false lorsque la balle est jetée (action Throw). Nous nous en servirons pour ne permettre qu'un unique lancé de la balle en asservissant l'affichage du menu à cette variable.

```
#pragma strict

/* Une variable booléenne donne l'état, en main ou non, de la boule */
private var bGrasped:boolean = false;

/*Une variable booléenne détermine si la balle est "disponible" (pas encore jetée) */
private var bAvailable:boolean = true;

/*Fonctions d'accès aux membres privés */
function isGrasped():boolean{return bGrasped;}
function isAvailable():boolean{return bAvailable;}

function Grasp()
{
    if(!bGrasped)
    {

    /*Mettre ici le comportement de Grasp*/

        bGrasped = true;
    }
}

function Release()
{

    if(bGrasped)
    {
        /*Mettre ici le comportement de Release*/

        bGrasped = false;
    }

}


function Throw()
{
    if(bGrasped)
    {
        /*Mettre ici le comportement de Throw*/


        bGrasped = false;
        bAvailable = false;
    }
}
```

Figure 1 : BallBehavior

Ajout d'une interface utilisateur

Créer un nouveau script "BallGUI" que vous veillerez à attacher à votre boule.
En ajoutant à ce script une fonction *OnGUI*(), on déclare la création d'une GUI (Graphical User Inteface) sur cet objet. Tout comme la fonction *Update*(), la fonction *OnGUI* () est appelée à chaque pas de simulation.
Dans cette fonction, nous allons déclarer des boutons en utilisant le scripting de GUI.

```
#pragma strict


private var ReleaseButton:Rect;
private var ThrowButton:Rect;
private var GraspButton:Rect;

function OnGUI()
{
    // The object is viewed by the main camera
    if(renderer.isVisible)
    {

        /*Retrieve another component of the same gameObject */
        var ballBehavior : BallBehavior = GetComponent(BallBehavior);


        // The object is available i.e. it has not been thrown before
        if(ballBehavior.isAvailable())
        {
            // Retrieve position of object in camera (in pixels !)
            var newpos : Vector2 = Camera.main.WorldToScreenPoint(transform.position);


            // Display the name at the retrieved position
            GUI.Label(Rect(newpos.x,Camera.main.GetScreenHeight()-newpos.y,30,30),name);
            GraspButton = Rect(newpos.x, Camera.main.GetScreenHeight()-newpos.y+40,50,30);


            // Verify if player can grasp this object by accessing its behavior component
            if(ballBehavior.isGrasped())
            {
                /*Affichage du bouton "Release"*/
                if(GUI.Button(ReleaseButton,"Release")) /* Implicit : if the button is triggered */
                {
                    ballBehavior.Release();
                }
                if(GUI.Button(ThrowButton,"Throw"))
                {
                    ballBehavior.Throw();

                }
            }
            else
            if(GUI.Button(GraspButton,"Grasp"))
            {

            /* COmpute and save Button position for Release and Throw according to current position of the ball */
                ReleaseButton = Rect(newpos.x-60, Camera.main.GetScreenHeight()-newpos.y+40,50,30);
                ThrowButton = Rect(newpos.x, Camera.main.GetScreenHeight()-newpos.y+40,50,30);
                ballBehavior.Grasp();
            }
        }
    }
}
```

Figure 2: Script BallGUI

Comme vous pouvez le constater, la déclaration du bouton et son utilisation se font en même temps dans une structure conditionnelle if(...).
Par ailleurs, nous voyons ici une première méthode d'appel à d'autres composants par l'utilisation de la méthode *GetComponent(NomDuComposant).*

Mise en place du jeu

Dans les assets fournis avec le package, vous trouverez un prefab *p_tenpin*. Etudiez ce prefab: quels sont ces composants? qu'est ce que cela implique?
Ajouter plusieurs instances du tenpin dans la scène en rang d'oignon. Veillez à ce que ces objets ne se touchent pas.

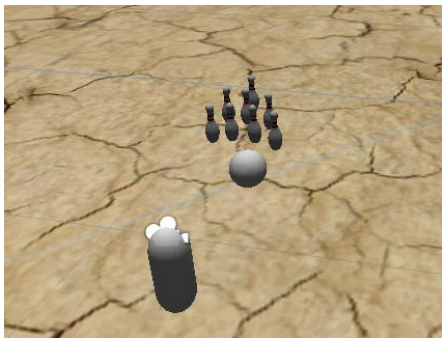

Figure 3 : Scène de bowling

Nous allons maintenant implémenter la gestion des quilles. Pour rappel, nous voulons compter les chocs entre les quilles.
Créer un script *TenpinControl* que vous veillerez à ajouter **au préfab** afin que cette modification soit reportée sur toutes les instances de tenpin.
Naturellement nous utilisons la fonction *OnColliderEvent*() pour détecter la collision de la quille avec un autre objet. Lors de la collision, nous utilisons un nouveau mécanisme de communication entre script : les messages.
Son utilisation est relativement simple, on envoie une chaine de caractère qui correspond au nom d'une fonction qui s'exécutera parmis les destinataires du *SendMessage*.
Dans le cadre de *SendMessageUpward*(), ces destinataires sont tous les éléments hiérarchiquement supérieurs à l'objet courant. La version *SendMessage*() envoie aux éléments hiérarchiquement inférieurs.
Notre objectif est de compter **uniquement** les collisions avec les autres quilles. Il faut donc distinguer quels sont les quilles parmi les objets collidant avec la boule. Il existe dans Unity un système de "tag" qui permet de classer les objets dans différentes catégories.
Dans l'*inspector* du préfab, créez un nouveau tag "Tenpin". Vérifiez que le tag est bien appliqué à toutes vos instances du tenpin.
L'attribut "tag" des GameObject peut être testé dans une structure conditionnelle afin de connaitre la catégorie d'un objet.

```
#pragma strict

function OnCollisionEnter(collideEvent:Collision)
{
    if(collideEvent.gameObject.tag == "Tenpin")
        SendMessageUpwards("FallenTenpin");
    }
}
```

Figure 4 : TenpinControl

Gestion du jeu : le GameManager

Nous allons mettre en place un GameManager rudimentaire qui stockera et mettra à jour le score.
Ajouter un script, appelé *GameManager*, au plan. Ce script conservera la valeur d'une variable entière *score* et interceptera le message "FallenTenpin" dans une fonction *FallenTenpin*() qui incrémentera le score.

```
#pragma strict

public var score:int;

function Start () {
    //Initialise score to 0 at the beginning of the game
    score = 0;
}


function FallenTenpin()
{
    score++;
}
```

Figure 5 : GameManager

Enrichissons maintenant l'interface du jeu afin d'afficher le score.
Créer une nouveau script *GameManagerInfoGUI* que vous associerez au plan avec le code suivant.

```
#pragma strict


function OnGUI()
{
    var gm : GameManager = GetComponent("GameManager");
    GUI.Label(new Rect(10,10,60,20),"Score : " + gm.score);
}
```

Figure 6 : GameManagerInfoGUI

Voici une nouvelle fois une utilisation de la communication entre component pour récupérer la valeur d'une variable.

Figure 7: Menu et score

Création d'un menu de lancement et gestion de scène

Nous allons créer une scène dédiée à l'affichage d'un menu. Ce menu comportera un bouton "Play" qui permettra de lancer la scène que nous avons précédemment créer et un bouton "Quit" qui permettra de fermer le jeu.
Avant cela, il est nécessaire d'ajouter la scène de jeu dans les Build Settings (Fichier -> Build Settings -> Add current scene).
Créer maintenant une nouvelle scène. Ajouter à la camera un nouveau script MenuGUI.

```
#pragma strict


function OnGUI()
{
    /* Size of buttons */
    var widthButton : int = 100;
    var heightButton:int = 20;


    /*Compute middle of the screen */
    var x:int = Screen.width/2-widthButton/2;
    var y:int = Screen.height/2;


    GUI.Label(new Rect(x,y,widthButton,heightButton),"MENU !");

    if(GUI.Button(new Rect(x,y+20,widthButton,heightButton),"PLAY!"))
        Application.LoadLevel("TD5"); /*Load your scene*/

    if(GUI.Button(new Rect(x,y+40,widthButton,heightButton),"QUIT"))
        Application.Quit();

}
```

Figure 8 : MenuGUI

Nous découvrons ici la classe **Screen** qui permet de disposer d'information et de fonctionnalité sur la surface d'affichage du jeu. Ici, on l'utilise pour positionner nos boutons au centre de l'image quelque soit sa taille.
Notez également la classe **Application** qui permet d'avoir différents contrôle sur l'application elle-même. Il est normal que la méthode *Quit*() ne semble pas fonctionner lorsque vous tester votre application dans Unity. Cette méthode est faite pour quitter l'application lorsque celle-ci est déployée en "Standalone" (sans Unity).

Figure 9 : Affichage du menu