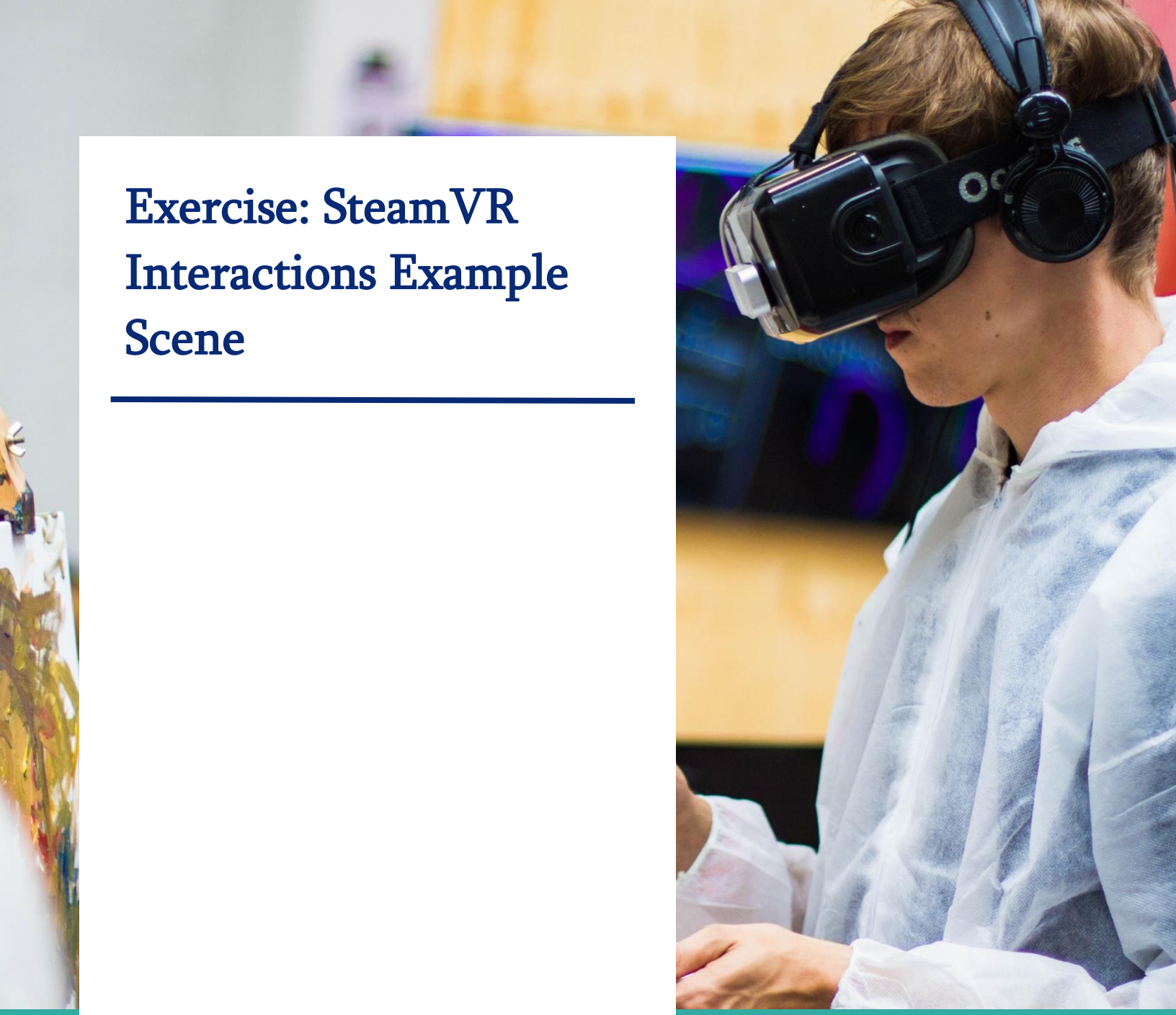

Exercise: SteamVR Interactions Example Scene

APRIL 24

Videogames and Virtual Reality

Authored by: Pedro Gómez López

Index of contents

Index of figures

Index of tables

Introduction

Virtual reality (VR) represents another way of interaction between the user and a specific environment, in which some technological components take special importance (i.e., powerful computer, VR viewers, head-mounted displays, head-phones, motion-sensing gloves, etc.). This document consists on a tutorial whose main intention is to put the user in contact with some specific technologies related with Virtual Reality. In particular, we will focus on SteamVR.

SteamVR is a virtual reality system, with a headset built by Valve and partner HTC and the headset itself is called the HTC Vive. Similar to what happens with Oculus Rift, the Vive is a VR device that contains two screens (1080x1200 resolution) streaming data at high refresh rates (90Hz) to create the sense of 3D virtual reality. In a common scenario, the headset will connect via a hardwire to a PC, which will run games and other VR software. However, it is also possible to test the different available possibilities within a simulator.

The main intention of this document is to put the user in contact with the SteamVR Interaction System, which is included in the SteamVR asset package for Unity. This objective is intended to be tackled by developing two main tasks:

- Break down one of the examples that are available in this Interaction System, describing it in a deep way. In our case, we will focus on the Throwables example.
- Once the example has been studied and split into smaller pieces of knowledge, we will create a very simple mini-game in which we are going to extend the functionality of the straightforward mentioned example.

Set up the environment

Hardware requirements

In order to be able to test in a real environment all the examples that are going to be described below, you should have at your disposal:

- **SteamVR/OSVR-compatible Head-Mounted Display (HMD)**, such as the **HTC Vive** and **HDK 1.4** used in this course. The HMD must be connected to the computer and SteamVR running prior to play the scenes.

Software requirements

In order to be able to test in a real environment all the examples that are going to be described below, you should have at your disposal:

- **Unity 2017.1.0f3**. You can download Unity from https://unity3d.com/es/get-unity/download.
- **Steam** by Valve, **SteamVR Plugin** for Unity and **OSVR server.**

First tutorial

The instructions to be followed in order to correctly installed all the necessary components that take part of the project's core are described in another tutorial (Molina Massó, 2018) (requires authentication). Consequently, it is highly recommended to firstly read and develop this tutorial as a previous step to be tackled before carrying out this tutorial.

In order to summarise it in a few words, what we are going to achieve once we finish the previous tutorial is to get a created project in Unity with the SteamVR Plugin downloaded from the Asset Store as a unity package that we will include in our project.

Interactions_Example scene

If you have reached this point it means that you have successfully completed the previous tutorial (Molina Massó, 2018). Perfect, ¡congratulations! Now, we are going to focus on the **Interactions_Examples** scene.

If you are using HTC Vive, it is a quite advisable scene to be played and tested. It features a whole host of examples for using with the Vive controllers for UI, teleportation, and interfaces. Maybe one of the most noteworthy example is the one that involves the longbow with the arrows. It consists on a longbow ready to use, complete with loading in arrows manually and firing them at a target.

In order to open the mentioned scene, you should follow these steps:

1. In the Project Panel, expand the **SteamVR** folder.
2. Open the scene named **Interactions_Example**, located in **Assets/SteamVR/InteractionSystem/Samples/Scenes** (Figure 1).
3. Press the **Play** button so that you can try out the scene.

Figure 1. Interactions_Example in Unity

Components

The elements that take part of this scene are reflected in the **Hierarchy** Panel.
As a first approximation to all of them, we could establish a classification:

- Core Elements. In this group we could include the **Player** component, and the **Environment** component. It may be controversial, but we could also include in this part the **Teleport** component, in the sense that it let us access the rest of examples.

- **Examples elements**. In this part we could include the rest of components or elements that take part in the scene, it is said, **UI&Hints**, **Throwables**, **InteractableExample**, **LinearDrive**, **CircularDrive** and **Longbow**.

In order to make use of a more specific language, when we talk about components or elements, we should use the word "GameObject", as it represents in a better and more accurate way that we are working on Unity.

In this sense, the **Player** GameObject represents the player in reality. All the elements associated with it are a set of scripts that make possible the functionality. The **Environment** GameObject represent the floor and walls that surround the scene, as well as all the lights and effects. However, the GameObject that is going to be the object of our studies is **Throwables**, and so we are going to focus our efforts on it.

Throwables GameObject

This is the main GameObject we are going to work with. There is a set of other GameObjects that are pending from it. The example comes with these ones[1]:

1. **Pedestal**. This is the structure that supports the set of spheres and cubes. It is composed of:
    a. **Cube (Mesh Filter)**. In this part, the Mesh attribute is set to **Cube**.
    b. **Box Collider**. We need it to be active so that spheres and cubes can actually stay on the pedestal.
    c. **Mesh Renderer**. This component adds some aspect attributes to the pedestal.

2. **TeleportPoint**. This GameObject contains two components, an animation and a script. The main intention of this GameObject is to give the user the possibility to be teleported just in front of the pedestal when playing the game with the real HTC Vive.

3. **TitleCanvas**. This GameObject contains two scripts and a Canvas Component. All these components are there in order to represent the text that is placed above the pedestal to inform about the game type. However, it is more intuitive the **Text** GameObject that pends from it. Inside this latter GameObject we can find the **Text (Script)** component. At this component is where we can actually change the text content and display characteristics.

4. **ThrowableBall**. There are up to two GameObjects of this type. It is composed of A **Rigidbody** component and some scripts that give the object functionality, Velocity Estimator, Interactable, Throwable and Interactable Hover Events, which come directly from SteamVR asset. Pending from it, there is another GameObject, which in each case is called **Sphere**. This GameObject consists on a **Sphere (Mesh Filter)**, a **Sphere Collider** and a **Mesh Rendered**.

[1] It is not going to be mentioned the Transform component, as it takes part of all GameObjects. However, it is useful for the user to take it into account.

5. **ThrowableCube**. There are up to five GameObjects of this type. It is composed of A **Rigidbody** component and some scripts that give the object functionality, Velocity Estimator, Interactable, Throwable and Interactable Hover Events, which come directly from SteamVR asset. Pending from it, there is another GameObject, which in each case is called **Cube**. This GameObject consists on a **Cube (Mesh Filter)**, a **Box Collider** and a **Mesh Rendered**.

It is important to have a good knowledge about this structure as long as it supposes the basis of the mini-game that is going to be implemented in the following section.

Pedro's Circus Game

Cool! If you have reached this point it means that you are really engaged, and you want to implement your own mini-game based on the Interactions_Example scene in Unity. This section is going to be divided in three main sections. Firstly, we are going to briefly describe the game that we want to implement. Secondly, we are going to include some new stuff to our scene, so that we can add functionality. Finally, once we have all the necessary tools, we are going to modify everything we need so that we can build our own circus game.

Game description

The game consists of 5 cubes and 3 balls of different colours (blue, white, red and yellow). All of them are placed on a surface (pedestal) which is divided into 4 different squares, each one with a different colour (blue, white, red and yellow). The objects colour have three states:

1. Before we put the pointer over them for the first time. They are coloured with multicolour lines.
2. After we put the pointer over them for the first time. They are coloured in blue, white, red or yellow.
3. When we put the pointer over them (without grabbing them). They are coloured in blue, white, red or yellow.

The thing is that at the beginning of the game all objects are in state 1. However, after that, depending on the object, they will have a colour or another in state 2. The point is that for a given object, colour in state 2 never is the same as the colour in state 3 for the same object.

The user is asked to order objects in each square according to their state 2 colour. It is a bit trickier as long as the user is only able to see colour at state 2 while the pointer is over that object. In order to have a better idea of what I am talking, I will include a video called *FinalState*.

Include extra components

Step 1. Add an extra ball

We need eight objects, taking into consideration balls and squares and so we have to define a new **ThrowableBall**. In order to do that, we have to be placed on the Hierarchy Panel, over one of the ThrowableBall GameObjects that are already available. Placed here, we press Ctrl + C and then Ctrl + V. An extra ThrowableBall will be created with the name **ThrowableBall (1)**. Click on it and change the name attribute to **ThrowableBall**. At the end, you should have should have something like Figure 2.

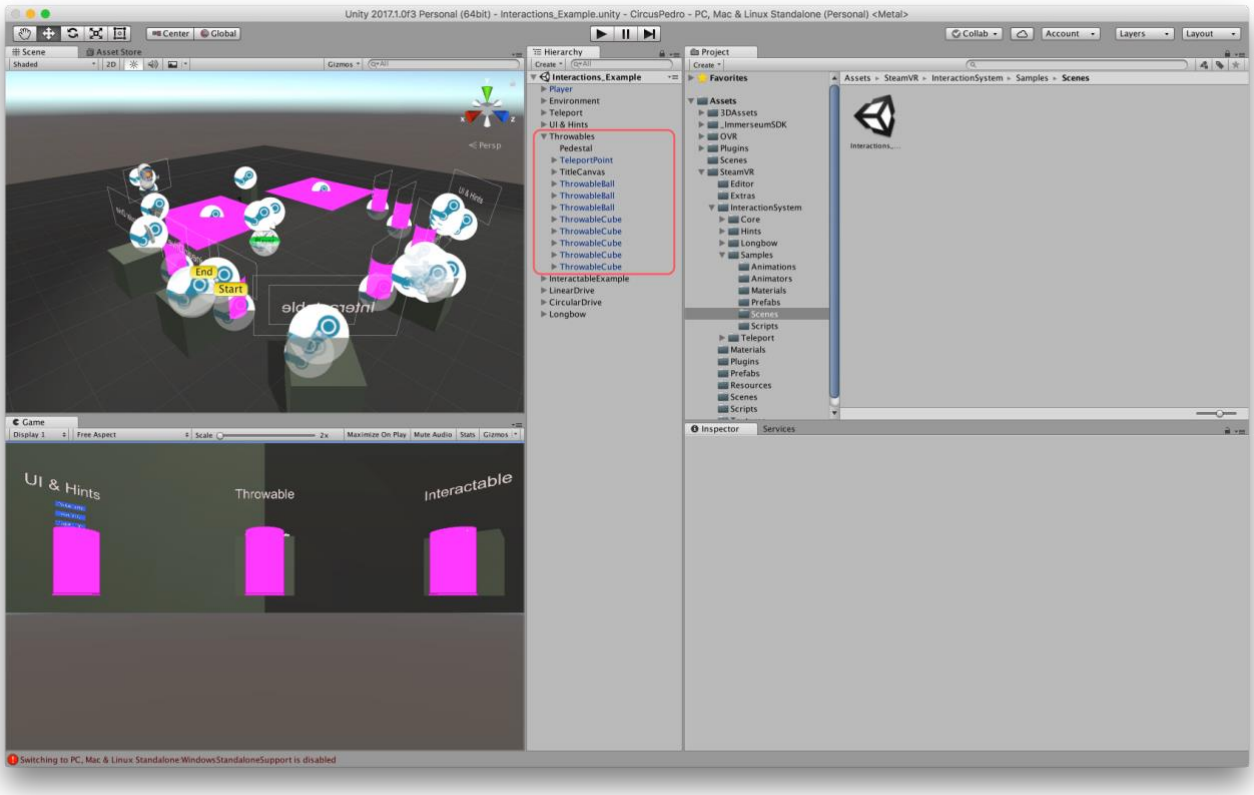

Figure 2. Project skeleton after adding the extra ThrowableBall

Step 2. Add an audio clip

Once the user is located in the **Throwables** GameObject, right click and select **Audio/AudioSource.** We will rename the GameObject to **Circus** and we will perform the following changes:

1. Download the audio file from this YouTube video: https://www.youtube.com/watch?v=ls1gnCHr14Q[2]
2. Drag and Drop the MP3 audio file from the folder you have it to the Assets folder in Unity.
3. Click on the circle which is next to the **AudioClip** attribute and select your MP3 file from the Assets Folder.
4. Check the **Play on Awake** attribute.
5. Check the **Loop** attribute.

At the end, everything should look like in Figure 3.

[2] You can put the music you want. It is just that I like that song ☺

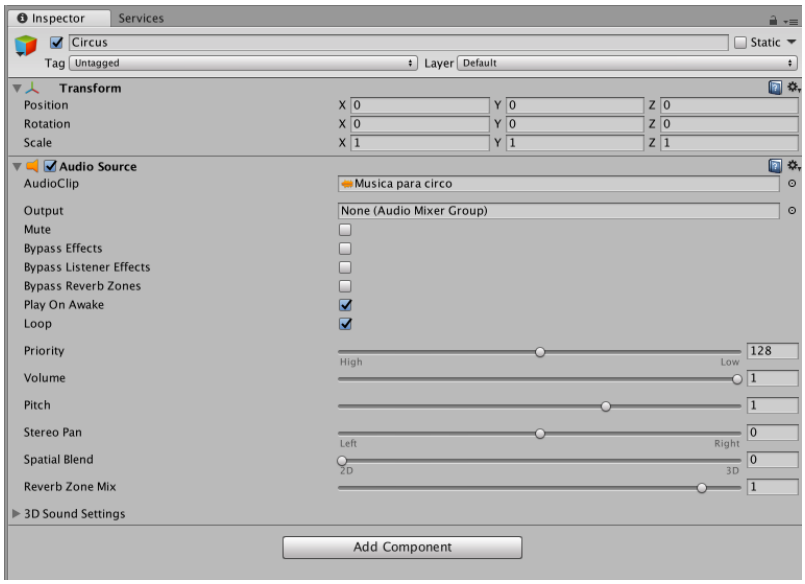

Figure 3. AudioSource GameObject

Step 3. Add the blue panel over the pedestal surface

Once the user is located in the **Throwables** GameObject, right click and select **3D Object/Plane.** We will rename the GameObject to **Plane_Blue** and we will perform the following changes:

1. Uncheck the **Box Collider** component.
2. Set **Transform** component attributes with these values:
    a. **Position. X:-0.25 Y:1.01 Z:0.25**
    b. **Rotation. X:0 Y:0 Z:0**
    c. **Scale. X:0.05 Y:0.05 Z:0.05**
3. Change attribute **Materials/Element 0** from the **Mesh Renderer** component. You can do this by clicking on the circle next to the attribute and selecting **ShinyWhiteHighlighted**.

Step 4. Add the yellow panel over the pedestal surface

Once the user is located in the **Throwables** GameObject, right click and select **3D Object/Plane.** We will rename the GameObject to **Plane_Yellow** and we will perform the following changes:

1. Uncheck the **Box Collider** component.
2. Set **Transform** component attributes with these values:
    a. **Position. X:0.25 Y:1.01 Z:0.25**
    b. **Rotation. X:0 Y:0 Z:0**
    c. **Scale. X:0.05 Y:0.05 Z:0.05**
3. Change attribute **Materials/Element 0** from the **Mesh Renderer** component. You can do this by clicking on the circle next to the attribute and selecting **ShinyYellow**.

Step 5. Add the white panel over the pedestal surface

Once the user is located in the **Throwables** GameObject, right click and select **3D Object/Plane.** We will rename the GameObject to **Plane_White** and we will perform the following changes:

1. Uncheck the **Box Collider** component.
2. Set **Transform** component attributes with these values:
    a. **Position. X:-0.25 Y:1.01 Z:-0.25**
    b. **Rotation. X:0 Y:0 Z:0**
    c. **Scale. X:0.05 Y:0.05 Z:0.05**
3. Change attribute **Materials/Element 0** from the **Mesh Renderer** component. You can do this by clicking on the circle next to the attribute and selecting **ShinyWhite**.

Step 6. Add the red panel over the pedestal surface

Once the user is located in the **Throwables** GameObject, right click and select **3D Object/Plane.** We will rename the GameObject to **Plane_Red** and we will perform the following changes:

1. Uncheck the **Box Collider** component.
2. Set **Transform** component attributes with these values:
    a. **Position. X:0.25 Y:1.01 Z:-0.25**
    b. **Rotation. X:0 Y:0 Z:0**
    c. **Scale. X:0.05 Y:0.05 Z:0.05**
3. Change attribute **Materials/Element 0** from the **Mesh Renderer** component. You can do this by clicking on the circle next to the attribute and selecting **Default_RED**.

Step 7. Add the functionality by creating a new script

In order to add the game some functionality, we will create a new Script. We will have to select from the Unity editor **Assets/Create/C# Script**. We will call to this script **PedroCircus.cs**.

Then, once the user has selected the **Throwables** GameObject, left click in **Add Component** and select **Script/CircusPedro.cs.** I will facilitate the script itself. It basically consists of two methods:

1. **Start()**. It is executed at the beginning.
2. **Update()**. It is executed one time per frame. What we basically do is to control the objects (spheres and cubes) position. Once a colour surface is completed the user is informed via Debug Message. Once all objects are located in the right place the music stops and it means that you have successfully reached the end of the game.

Modify components

Step 1. Change the Canvas title

In order to do this, we will modify the GameObject **TitleCanvas/Text**. We will set the **Text** attribute of the **Text (Script)** component to “Pedro’s Mind Circus” as shown in Figure 4.

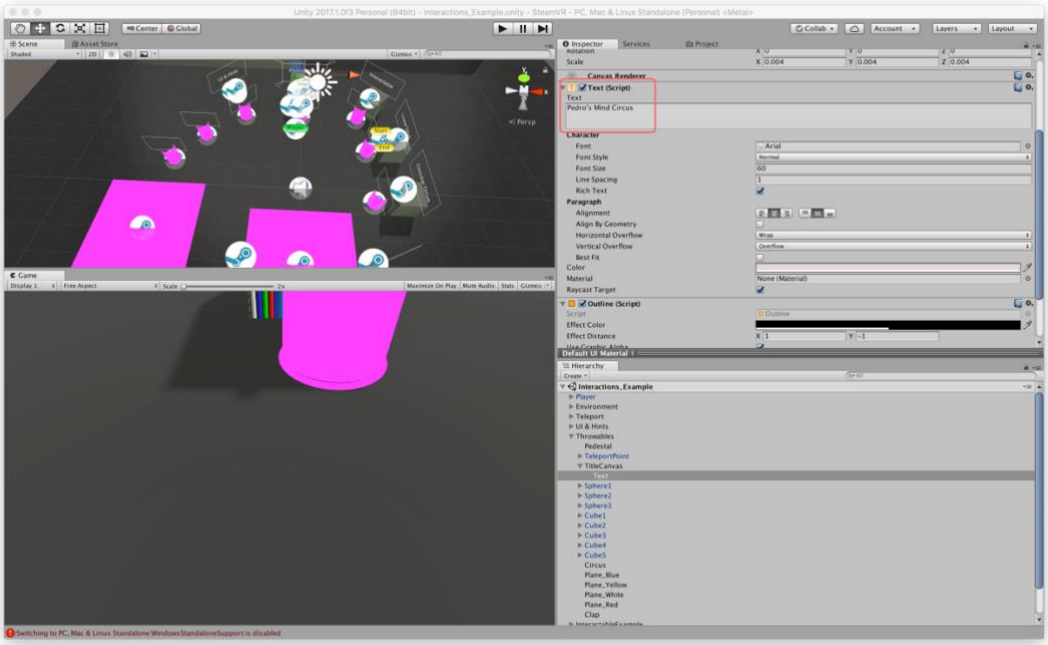

Figure 4. Change Canvas title

Step 2. Modify information showed on the screen

This is the information that is fixed in the screen at the left-hand side. In order to change this you have to perform some changes in a script called **FallbackCameraController.cs.** This script is located in Project Panel, specifically in **Assets/SteamVR/InteractionSystem/Core/Scripts/ FallbackCameraController.cs.** In particular, you have to change the **OnGUI()** method so that it looks like in Figure 5.

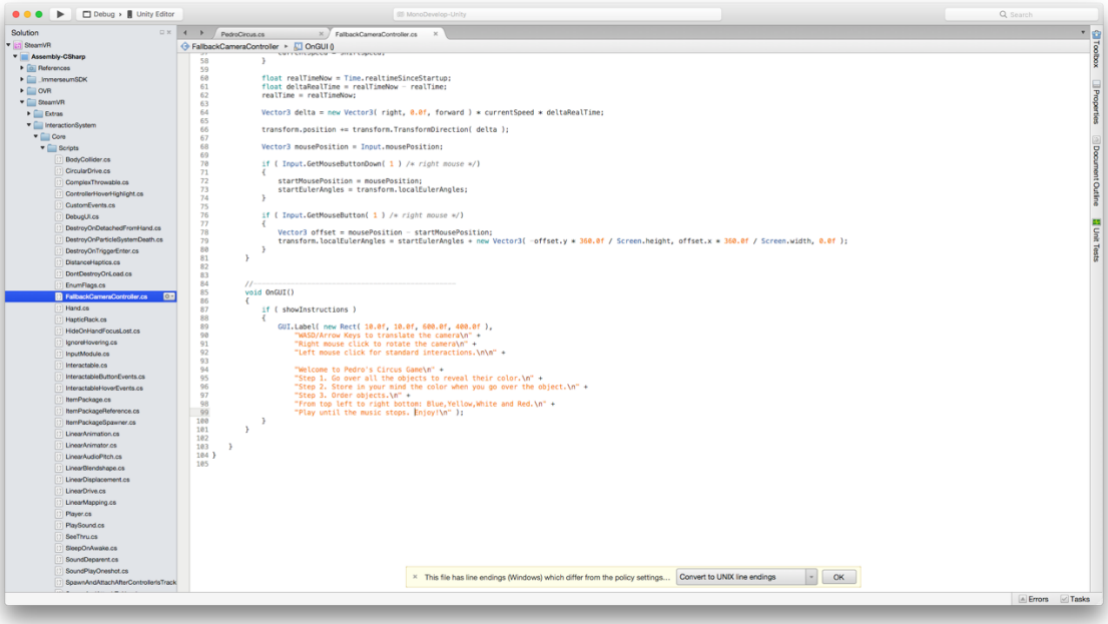

Figure 5. Change displayed text on screen

Step 3. Modify GameObjects names

We want things to be as clear as possible. Consequently, we will rename the following GameObjects:

1. **ThrowableBall** to **Sphere1**.
2. **ThrowableBall** to **Sphere2**.
3. **ThrowableBall** to **Sphere3**.
4. **ThrowableCube** to **Cube1**.
5. **ThrowableCube** to **Cube2**.
6. **ThrowableCube** to **Cube3**.
7. **ThrowableCube** to **Cube4**.
8. **ThrowableCube** to **Cube5**.

It is quite important to correctly perform this step as long as the script that gives the game functionality consumes them in a correct way only if their names are specified as above.

Step 4. Set objects and pedestal colour to state 1

In order to do this, we have to change the **Mesh Renderer** component of all the children pending from **Sphere1**, **Sphere2**, **Sphere3**, **Cube1**, **Cube2**, **Cube3**, **Cube4** and **Cube5**. We will also change the **Pedestal**. So, in the **Mesh Renderer** component, we have to put the **Materials/Element 0** attribute to **palette** (material with the multiline colours).

Step 5. Set spheres colour properties

It is necessary to assign to the spheres different colours depending on their state. Assuming that we have already performed Step 4, we are now going to set parameters for states 2 and 3. In order to do so, we need to have selected the GameObject **SphereX,** and then we have to change the component **Interactable Hover Events (Script)**, particularly the parameters of the functions **On Hand Hover Begin ()** and **On Hand Hover End (),** and more specifically, the parameter **MeshRenderer.material.** In Table 1 it is reflected the values that these parameters should take for each sphere object.

Table 1. Spheres colours

GameObject	On Hand Hover Begin ( ) MeshRenderer.material	On Hand Hover End ( ) MeshRenderer.material
Sphere1	ShinyYellow	ShinyWhiteHighlighted
Sphere2	ShinyWhiteHighlighted	ShinyYellow
Sphere3	ShinyWhite	ShinyWhiteHighlighted

Step 6. Set cubes colour properties

Exactly the same as we have done in Step 5, it is necessary to assign to the cubes different colours depending on their state. Assuming that we have already performed Step 4, we are now going to set parameters for states 2 and 3. In order to do so, we need to have selected the GameObject **CubeX,** and then we have to change the component **Interactable Hover Events (Script)**, particularly the parameters of the functions **On Hand Hover**

Begin () and **On Hand Hover End (),** and more specifically, the parameter **MeshRenderer.material.** In Table 2 it is reflected the values that these parameters should take for each cube object.

Table 2. Cubes colours

GameObject	On Hand Hover Begin ( ) MeshRenderer.material	On Hand Hover End ( ) MeshRenderer.material
Cube1	ShinyWhiteHighlighted	Default_RED
Cube2	ShinyYellow	ShinyWhite
Cube3	Default_RED	ShinyYellow
Cube4	Default_RED	ShinyWhite
Cube5	ShinyWhite	ShinyWhiteHighlighted

Step 7. Modify Player camera

The original scene has the camera set by default to a particular position. Now, we are going to change it so that the initial player view is more accurate. So, in order to do that we are going to modify the **Transform** component of the **Player** GameObject and we will set the attributes values as here:

a. **Position. X:-0.3 Y:0.9 Z:2.5**
b. **Rotation. X:50 Y:0 Z:0**
c. **Scale. X:1 Y:1 Z:1**

Once we have changed this parameters the game should look like when starting it.

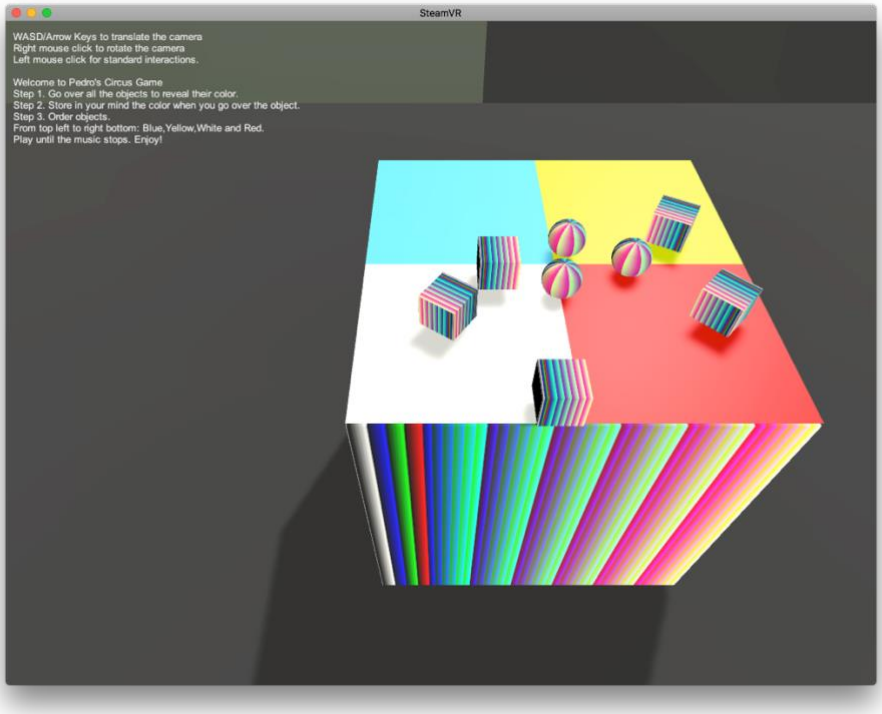

Figure 6. Game initial view

Step 8. Brief explanation about PedroCircus.cs script

This file gives the game the functionality to be playable. In order to achieve it, we are going to explain the main concepts. As we have already mentioned, there are two methods in it, **Start()** and **Update()**. Apart from these methods, we have also defined some variables:

```
1.
2.    public  string  myName;
3.    GameObject  blueP,  whiteP,  redP,  yellowP;
4.    GameObject  sphere1,  sphere2,  sphere3;
5.    GameObject  cube1,  cube2,  cube3,  cube4,  cube5;
6.    GameObject  circus;
7.    bool  yellow  =  false;
8.    bool  blue  =  false;
9.    bool  red  =  false;
10.   bool  white  =  false;
```

These variables are useful for assigning them the different GameObjects we are going to work with. Then, there is the **Start( )** code:

```
1.    // Use this for initialization
2.    void Start() {
3.        Debug.Log("I am alive! and my creator is " + myName);
4.        sphere1 = GameObject.Find("Sphere1");
5.        sphere2 = GameObject.Find("Sphere2");
6.        sphere3 = GameObject.Find("Sphere3");
7.        cube1 = GameObject.Find("Cube1");
8.        cube2 = GameObject.Find("Cube2");
9.        cube3 = GameObject.Find("Cube3");
10.       cube4 = GameObject.Find("Cube4");
11.       cube5 = GameObject.Find("Cube5");
12.       blueP = GameObject.Find("Plane_Blue");
13.       whiteP = GameObject.Find("Plane_White");
14.       redP = GameObject.Find("Plane_Red");
15.       yellowP = GameObject.Find("Plane_Yellow");
16.       Debug.Log("Information about the objects");
17.       Debug.Log("Sphere 1 position is: " + sphere1.transform.position);
18.       Debug.Log("Sphere 2 position is: " + sphere2.transform.position);
19.       Debug.Log("Sphere 3 position is: " + sphere3.transform.position);
20.       Debug.Log("Cube 1 position is: " + cube1.transform.position);
21.       Debug.Log("Cube 2 position is: " + cube2.transform.position);
22.       Debug.Log("Cube 3 position is: " + cube3.transform.position);
23.       Debug.Log("Cube 4 position is: " + cube4.transform.position);
24.       Debug.Log("Cube 5 position is: " + cube5.transform.position); //Panel information
25.       Debug.Log("Information about the panels");
26.       Debug.Log("Blue Panel position is: " + blueP.transform.position);
27.       Debug.Log("White Panel position is: " + whiteP.transform.position);
28.       Debug.Log("Red Panel position is: " + redP.transform.position);
29.       Debug.Log("Yellow Panel position is: " + yellowP.transform.position);
30.   }
```

As you can see, what we do in here is just locating the objects and print some useful information for the developer. This information is printed at Runtime in the console, so it is transparent for the final user.

And finally, we have the **Update( )** code:

```
1.    // Update is called once per frame
2.    void Update() { // Yellow zone
3.        if (!yellow) {
4.            if ((sphere1.transform.position.x >= (0.1 f) && sphere1.transform.position.x <= (0.5
f)) && (sphere1.transform.position.z >= (5.1 f) && sphere1.transform.position.z <= (5.5 f)) && (cube
2.transform.position.x >= (0.1 f) && cube2.transform.position.x <= (0.5 f)) && (cube2.transform.posi
tion.z >= (5.1 f) && cube2.transform.position.z <= (5.5 f))) {
5.                Debug.Log("You have completed the yellow zone!");
6.                yellow = true;
7.            }
8.        } // Red zone
9.        if (!red) {
10.           if ((cube3.transform.position.x >= (0.1 f) && cube3.transform.position.x <= (0.5 f))
&& (cube3.transform.position.z >= (4.6 f) && cube3.transform.position.z <= (5.0 f)) && (cube4.transf
orm.position.x >= (0.1 f) && cube4.transform.position.x <= (0.5 f)) && (cube4.transform.position.z >
= (4.6 f) && cube4.transform.position.z <= (5.0 f))) {
11.               Debug.Log("You have completed the red zone!");
12.               red = true;
13.           }
14.       } // Blue zone
15.       if (!blue) {
16.           if ((sphere2.transform.position.x >= (-0.5 f) && sphere2.transform.position.x <= (-
0.1 f)) && (sphere2.transform.position.z >= (5.1 f) && sphere2.transform.position.z <= (5.5 f)) && (
cube1.transform.position.x >= (-0.5 f) && cube1.transform.position.x <= (-
0.1 f)) && (cube1.transform.position.z >= (5.1 f) && cube1.transform.position.z <= (5.5 f))) {
17.               Debug.Log("You have completed the blue zone!");
18.               blue = true;
19.           }
20.       } // White zone
21.       if (!white) {
22.           if ((sphere3.transform.position.x >= (-0.5 f) && sphere3.transform.position.x <= (-
0.1 f)) && (sphere3.transform.position.z >= (4.6 f) && sphere3.transform.position.z <= (5.0 f)) && (
cube5.transform.position.x >= (-0.5 f) && cube5.transform.position.x <= (-
0.1 f)) && (cube5.transform.position.z >= (4.6 f) && cube5.transform.position.z <= (5.0 f))) {
23.               Debug.Log("You have completed the white zone!");
24.               white = true;
25.           }
26.       } // Once I have all zones completed I stop playing the sound
27.       if (red && yellow && blue && white) { // Add sound logic in here
28.           Debug.Log("You have completed the game!");
29.           circus = GameObject.Find("Circus");
30.           circus.GetComponent < AudioSource > ().Stop();
31.       }
32.   }
```

In this part, that is executed once per frame, we check each object position (taking into consideration that we know in advance their colour). The user is informed via console messages when each zone is complete. Finally, when all zones are correctly completed, and so the Boolean variables set to true, the music stops playing.

Final considerations

Pedro Circus Game is a very simple game that tries to improve user's knowledge about SteamVR Interactions_Example Scene. Obviously, there are many things that could be improve and many good ideas that could arise as a result of developing this project. There are much more resources associated with the project in author's Github account. Please, visit https://github.com/pegomez/Exercise1_Interactions-Example if you want further contents about this project. And in case you have any question, do not doubt on sending me an email to pegomez.lopez@gmail.com.

Bibliography

Molina Massó, J. P. (2018). Playing SteamVR example scenes in Unity. Retrieved April 24, 2018, from https://campusvirtual.uclm.es/pluginfile.php/2695555/mod_resource/content/1/Tutorial Playing SteamVR example scenes in Unity.pdf