VR Lab #3: Interaction

1.1. Introduction and goals

During the first two labs, you have learned how to reproduce content in virtual environments, and how to design simple, yet meaningful scenarios. This one will focus on *interaction*, a key aspect in VR that can noticeably increase immersion and realism, enhancing the overall experience. Actions like grabbing items, holding, or throwing them in a realistic way can drastically improve the final experience, and make it feel more realistic. This last laboratory assignment will also serve as a first contact with the setup you will be using in your Course Project.

In this assignment, we are going to apply basic knowledge about movement and interaction in virtual environments, and we will, for the first time, use an Head-mounted Display (HMD) and an external controller. Specifically, you will learn how to:

- Use open-source, pre-designed assets.
- Use Unity's physics engine, including collisions and forces.
- Port a desktop-like application to a VR setup.
- Use an HMD and a bluetooth controller to run an application (specifically, to play a game).

Equipment. We will be using an HMD based on a headset with a set of lenses, inside of which the user places their smartphone. Therefore, it is the smartphone that contributes the screen, the inertial measurement units (IMUs) used for tracking, and the computing power required for tracking, rendering and interaction. The HMD additionally has a bluetooth, handheld controller that will aid in the interaction.

There will be different types of tasks in this lab sessions. Most of them are small, self-explanatory tasks to give you a robust basis for the next assignments. When a new concept is introduced, you may have some **mandatory tasks** to reinforce the learning. Mandatory tasks are preceded by [MT-X]. Finally, **optional tasks** are presented at some points, and preceded by [0T-X]. To be eligible for continuous evaluation, you must complete at least all the mandatory tasks. To opt for the maximum grade (10/10), you need to complete both mandatory and optional tasks.

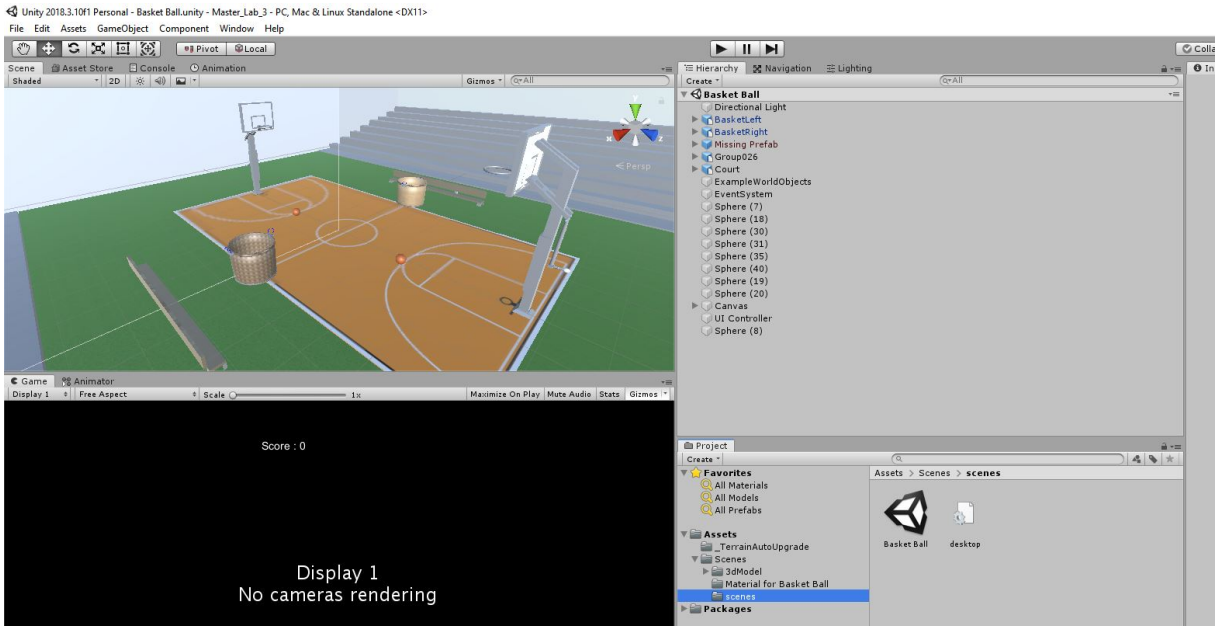

Fig. 1.1: The basketball playground already included in the project. The *Project* tab includes the unzipped files. *Scene* tab contains the playground correctly established.

1.2. Setting up the environment

First of all, we are going to set up the environment for this lab. Instead of manually creating our environment, we are going to leverage an open-source one. **Tip: In certain occasions, people develop their own projects and make them available for public use. This means that it is quite possible that your problem has been already tackled by someone (at least partially). It is thus a good practice not to reinvent the wheel, but to search for some similar projects and use them as your baseline. When doing this, always pay attention to the licenses, and give proper credit.** We will firstly download a simple, yet colorful and complete basketball court [1]. Download the repository to your computer, and unzip it. Create a new *3D project* in Unity as in previous labs, and keep track of where you created it. From the already downloaded file, go to the folder `3dBasketBall/Assets/` and move the folders `scenes, Material for Basket Ball`, and `3dModel` from the unzipped file to your project folder. Open Unity, and navigate to your *Project* tab, and then go to `Assets/Scenes/scenes/` (it should be there now that you moved the content from the .zip to your project). Find the scene called "Basket Ball" and open it. You will find your basketball court ready (see Figure 1.1).

1.3. Adding camera and player

Currently, there is neither camera nor player in the game. We will have to add them. To add the camera, right click in the *Hierarchy* tab and select camera. Make sure that your camera is *tagged* as *MainCamera*; otherwise, calling the variable "Camera.main" in the scripts would not

[1] Publicly available on https://github.com/sohail0992/basketBallGame

work (since there would be no main camera attached to your project). We will need to move our camera, so it is a good practice to create an empty object in top of the hierarchy (i.e., at the same level as the *Directional Light*), and then include the camera as a children of that empty object. To do this, create an empty object and rename it as "CameraParent", and place it in the correct position of the hierarchy. Then drag your new Camera object to it, and make sure that the camera has become its children.

We now want the camera to be centered, in case it is not. First of all, the camera coordinates must never be manipulated, so make sure its position and rotation coordinates are all 0. Then, go to the *CameraParent* and change its position to something similar to $(0, 1, 0)$, and fix its rotation to $(0, 0, 0)$. Make sure that you put your camera at a reasonable height: Too low or too high positions may cause you discomfort or nausea when playing the final VR experience. **Note: This is deliberate: Although this lab is done with a basic setup (HMD + Bluetooth Controller), when you implement an application with a more complex HMD (e.g., HTC Vive or Oculus Quest 2), you will not be able to manipulate the camera position directly, but only its parent**.

As we mentioned, we also need a player. To simplify this lab, we are going to assume that this game is played in first person, and therefore the camera accompanies the player. This means that the object *CameraParent* will also be the player.

1.4. Desktop interaction

Implementing an application directly in VR is not a good practice, since there are more degrees of freedom that may complicate the task. We are therefore implementing a simple desktop interaction first, and will then swap to virtual reality controls.

We will firstly create an script that allows our *CameraParent - our player* to move with the keyboard. Remember what you learned on previous labs, and **create a new script, write the corresponding code, and include it in your player object**. Make sure it works before stepping into the next task.

When you can move with your keyboard, add a functionality (similar to what you learned in Lab #1) that allows your player to "look" at the same direction of your mouse[2].

Now you can look and move around. We now want to catch a ball. We will let our player catch a ball when it gets closer to it. We then need to use Unity's **collision system**. This technique allows Unity to know when two objects are touching or colliding. The basket balls from the assets are already set up, but we still need to prepare our player to be able to interact with them.

Select *CameraParent* and go to *Inspector* tab. There, click on *Add Component* and add both a *Rigidbody* and a *Capsule Collider*. The former is included to tell Unity this object will interact with other scene elements. Make sure that *Use gravity* option is unchecked, and check *Is kinematic*. This way, our player functionality will remain correct (it will not fall through the

[2]Remember that simply moving forward is not enough. Your "forward" direction depends on your orientation. You have to move depending on your rotation. Hint 2: Remember that your y coordinate should never change.

floor, and will be affected by the gravity). On the other hand, the capsule collider will create an area around our player that will define when she is touching any object. Make sure - by pure visual inspection - that its radius and height is reasonable (e.g., $r = 1$ and $height = 3$. Also, uncheck *Is trigger* option.

We will then add a short code snippet that allows our player to catch the ball when it touches it. Go to the movement script you previously created and add the following variables:

```
bool isHolding = false;     // Is the player holding a ball?
GameObject item = null;     // Ball GameObject
float throwForce = 30.0f;   // Throwing force
Vector3 objectPos;          // Ball position
float distance;             // Distance between player and ball
```

We first need to know when our layer's collider and the ball's one touch. Unity has a very complete (and complex) physics engine with many functionalities already built in. We will leverage them, and add the following function. Be careful with this function. If your computer is too (too) fast, it may trigger the collision multiple times (meaning that, even though you have thrown the ball, it will collide again in the air with the player, and you will catch the ball before it is actually thrown. To avoid this behavior, you can add a timer in this function, such as in Lab#2, so that the condition can only be entered e.g., once per second. It is a good practice that you implement your application trying to foresee different bugs or undesired situations that can occur depending on different factors.

```
void OnCollisionEnter(Collision obj)
{
    if (obj.gameObject.tag == "Spheres")
    {
        item = obj.gameObject;
        item.transform.position = Camera.main.transform.position
                        + Camera.main.transform.forward * 0.75f;
        isHolding = true;
        item.GetComponent<Rigidbody>().useGravity = false;
        item.GetComponent<Rigidbody>().detectCollisions = true;
    }
}
```

The function *OnCollisionEnter* is triggered automatically when two colliders (remember to set the player's collider as aforementioned) touch. In this function, we will check whether our player has touched an *Sphere* object (basketball items are already tagged as spheres). If that is the case, we will update our touched item, we will "pick up"the ball (simulated by increasing its height a bit), we will update our "is holding"flag, and adjust gravity and kinematics attributes from the ball, so it does remain in our hands. This function is only called when our player firstly collides with the ball, so we need to keep updating the game while we are holding the ball. We will therefore add the following to our *Update* function:

```
    // Catch the ball
    if (item != null)
    {
        distance = Vector3.Distance(item.transform.position,
                                    transform.position);
        if (distance >= 1.5f)
        {
            isHolding = false;
        }
    }
```

We will first check whether we have a ball (if we have not picked up any, we can not compute any distance). If there is a ball picked up, we can calculate our distance to it, and decide whether we are still holding it or not.

```
    // Catch the ball
    if (item != null)
    {
        distance = Vector3.Distance(item.transform.position,
                                    transform.position);
        if (distance >= 1.5f)
        {
            isHolding = false;
        }
        if (isHolding == true)
        {
            item.GetComponent<Rigidbody>().velocity = Vector3.zero;
            item.GetComponent<Rigidbody>().angularVelocity = Vector3.zero;
            // The CameraParent becomes the parent of the ball too.
            item.transform.SetParent(transform);

            if (Input.GetKey("z"))
            {
                // Throw
            }
        }
        else
        {
            objectPos = item.transform.position;
            item.transform.SetParent(null);
            item.GetComponent<Rigidbody>().useGravity = true;
            item.transform.position = objectPos;

        }
}
```

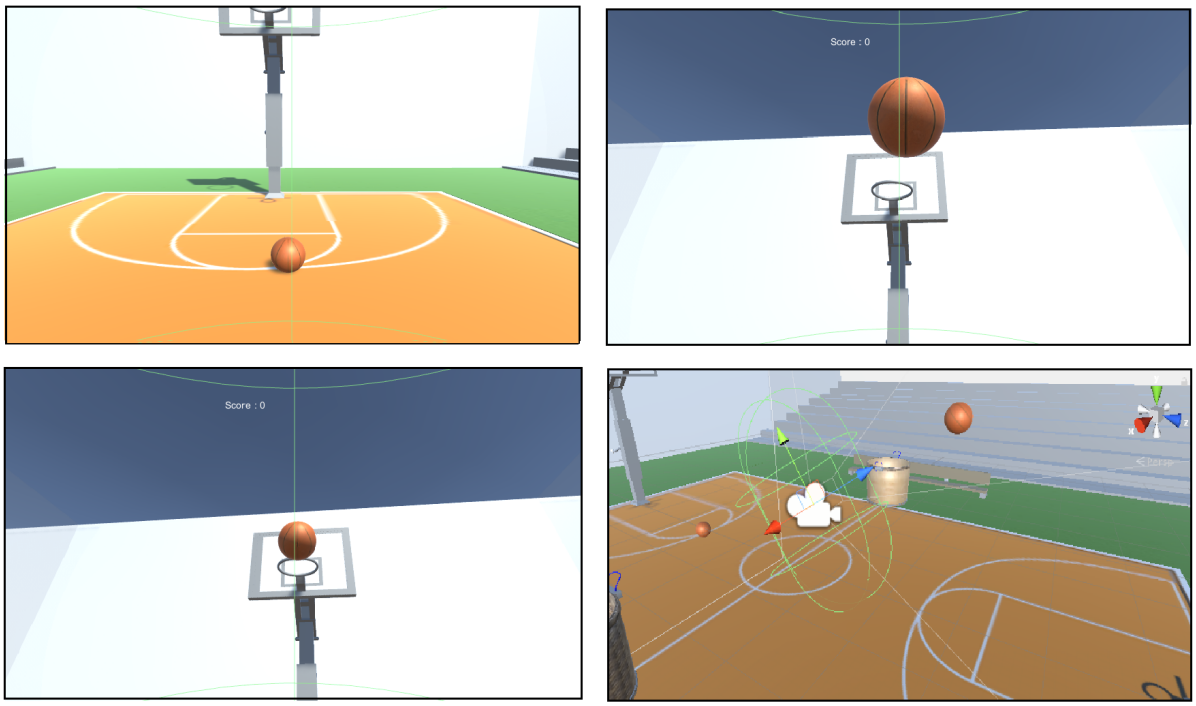

Fig. 1.2: Different stages of the throwing process. *Top-left:* The player standing in front of the ball. *Top-right:* The player holding the ball. *Bottom-Left:* The player has thrown the ball. *Bottom-right:* The *Scene* tab showing player's collider with green lines, whilst the ball is thrown to the basket.

If we are still holding a ball, we should eliminate its velocities, and indicate the game that we are now its parent (now the ball is another children like the camera). If we are not holding the ball anymore, we should remove all the changes we previously did, including reactivating its gravity and removing the player as its parent object, so that it can bounce away.

```
if (Input.GetKey("z"))
{
    // Throw
    var cam = Camera.main;
    item.GetComponent<Rigidbody>().AddForce(
                    cam.transform.forward * throwForce);
    isHolding = false;
}
```

We finally add a throw functionality. We basically aim at the same direction as the camera does, and add a force (again, Unity physics engine handles this on its own. If you find interesting how this engine works, do not hesitate on asking us). Of course, the player is no longer holding the ball, so we update that flag too.

We now have a game that allows us to throw a ball like in a basketball game (see Figure 1.2).

[MT-0] Add a new movement paradigm that instead of allowing to continuously move with

some keys, it allows you to directly—and instantly—teleport yourself a certain distance in a certain direction.

[MT-1] Now, instead of automatically teleporting to that position, change your implementation so that your first key input shows a red circle at the position you will teleport to, and with a second input, do teleport.

1.4.1. Switching to virtual reality

We have to update the way we move to work in virtual environments, only with our head and one controller (*This part will only require to adapt the first way of movement, i.e., the continuous movement.* Do not implement [MT-0] nor [MT-1] in VR). First, we need to make sure our game will correctly supports Joystick interaction. Go to *Edit* > *Project Settings* > *Input* in Unity. There, find the second appearance of "Horizontal" and "Vertical". Make sure the later has "Invert" checked, and that both of them have *Joystick Axis* as type, with its corresponding axis correctly selected.

Then, go to the movement script. We first need to remove both the translation and rotation snippets we included for the desktop version. Then, we may add the following:

```
if (Input.GetAxis("Vertical") > 0)
{
    transform.position += transform.forward * Time.deltaTime * speed;
}
else if (Input.GetAxis("Vertical") < 0)
{
    transform.position -= transform.forward * Time.deltaTime * speed;
}
else if (Input.GetAxis("Horizontal") < 0)
{
    transform.position -= transform.right * Time.deltaTime * speed;
}
else if (Input.GetAxis("Horizontal") > 0)
{
    transform.position += transform.right * Time.deltaTime * speed;
}

transform.position = new Vector3(
            transform.position.x,
            1.0f,
            transform.position.z);
```

With this code, our game will support movement when some input is done either to the horizontal axis or to the vertical axis. You may notice that we have not included any code for rotation. We will not need it: When we export our application (at the end of this section), we

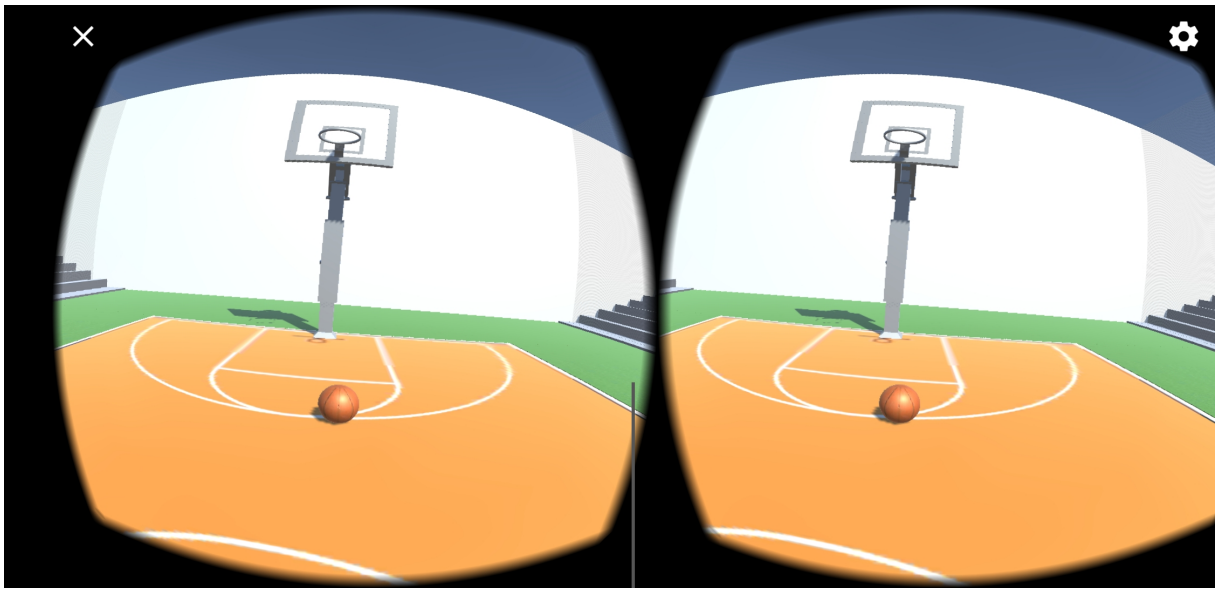

Fig. 1.3: Screenshot from the application running in a smartphone. With the smartphone correctly placed in the HMD, the experience will be identical to a standard HMD.

will indicate Unity it is a VR app designed for mobile, so it will make use of the gyroscope of the mobile to detect the correct orientation.

We must change the way we shoot the ball. This controllers are not as functional as Oculus's or HTC's controllers, but it allows our mobile to detect whether a button has been pressed. Replace your key detection with the following line:

```
if (Input.anyKey)
{
    // Throw the ball as before
}
```

We are now ready to move our project to virtual reality (VR). We firstly need to make sure that our computer has all the necessary functionalities. We need a stable version of Android Studio with its corresponding Android SDK. If we have it already installed, we will go to the *File* menu in Unity and select *Build Settings*. We will choose our platform (Android / iOS)[3], and click on *Switch Platform*. After that, we will configure the project in the same way as in Lab #1 and click on *Build*. **Important**: Please follow carefully all the instructions in Lab #1 to build the application into your smartphone. Missing a single step can prevent a successful build.

Note: You may need to have "Graphics API" selected on Android, and enable support for Vulkan.

Your APK will be exported to the folder you indicated, and you can then install it in your

[3]This lab has been tested only in Android devices. If you can only work on iOS and find any difficulty, please write us.

device. Activate your phone's bluetooth, pair your controller with your phone, select the correct mode of your controller (horizontal game mode, @ + B, see documentation), and then open your game (see Figure 1.3)[4]. Place your phone in the HMD, and enjoy your game.

[MT-2] Think of all you have learned in the Lectures, and explain how would you use an HMD with controllers that have tracking available to improve this game experience, and include a brief explanation in your report.

[OT-0] Search some information about the game *Half Life: Alyx*, and the different ways of interaction and movement it offers. This game has been reported to be the best VR game ever created, with incredibly high quality, and many interaction paradigms. Include some information and insights in your report.

1.5. Reporting your results

You should submit a **report** (in .PDF format) including your results for the tasks (results should be clearly labeled following the labels of the tasks they correspond to). There is no required style or fixed structure for your report, you will have to choose how to report the work you did. There are, however, some guidelines that you need to follow:

- The report should not be longer than **three pages** (*tres "carillas", no tres páginas*), and should at least address the mandatory tasks. The report file should be named `labXX-mainReport-YYYYYY.pdf`, with `XX` the number of the lab, and `YYYYYY` your NIP.
- Do not include whole snippets of code in the report: You may submit an additional .ZIP file with the code and shaders you wrote, and indicate in the report its function with a couple sentences at most.
- Including in the report the **main difficulties, thoughts, or insights** you had through the whole laboratory, as well as its relation to the lectures of the course, will be positively evaluated.
- If there is any other part of your work that cannot fit in the report (e.g., short videos or sets of a large number of screenshots), you should submit them in a separated, supplementary .ZIP file, **but always adequately referencing it in the main report, so we are aware of the existence of that file**.
- We will not look at submitted items that have not been referenced in the main report.

In terms of evaluation, mandatory tasks can get you up to 8 points out of 10, and the 2 remaining points can be obtained through the optional tasks and the report.

You should submit the assignment via Moodle, uploading them in the corresponding Moodle task. **Deadline: April 9th, 2024 @ 23:59**.

[4]Note that, in this image, the center is not totally centered. This happens when your mobile phone has a notch, but you use fullscreen mode. To correctly center your application, deactivate the fullscreen mode of your notch.