Universitat Pompeu Fabra
Miguel Casasnovas Bielsa

Extended Reality (XR) Game

Development Report

Introduction

For the final thesis, a basic interactive **XR application** was developed using **Unity** without previous experience with the development platform, the programming language used (C#), or extended reality.

This report details all the steps followed in the application's design, development, and deployment.

Constraints

The first part of the creative process was to produce an idea. To this end, ideas were brainstormed, and the constraints and requirements to exclude some of them were specified.

Constraints and requirements

- **Time Limitation**: the time to develop the game was limited. Thus, the game should be basic and not require a significant investment of time.
- **Difficulty level**: the complexity should be moderate or low, as it was individual work without extensive knowledge of video game development or the Unity platform.
- **Genre**: the game should be entertaining and educational.
- **Typology**: the game to be developed had to be an XR game. Since XR includes Virtual Reality (VR), Augmented Reality (AR), and Mixed Reality (MR), it was decided to develop a VR game.

Game description

In this section, the initial concepts of the video game are described. The game did not require a high level of complexity and was adequate to have only one level.

Game Name

Alteration Hunting

Game Story

There will be a collision between universes in the room!

Some items will be swapped with entities from another universe, and new items will appear. You must remember which objects are in the room to detect objects that do not correspond to your reality. Luckily, you have a portal gun to return them to their origins.

You must find and return these alterations to keep the multiverse stable!

Game operation

The player is in the room and can interact with the objects. There is limited time to remember the elements in the room and their colors before the multiverse collision occurs.

When time runs out, the room is altered. Fortunately, a portal gun was in the room and did not disappear. This pistol displays how many alterations have occurred. The user must identify objects that do not belong to the initial room and shoot them with the portal gun to return them to where they belong.

When the user shoots and hits an object correctly, it is returned to its original universe. If the user removes all disturbances, the world is saved, and the game is won. However, if an object that was not altered is shot at, it will cause a new disturbance in the multiverse: GAMEOVER.

Difficulty Level	Moderate
Genre	Educational, Shooter
Type	VR

Benefits

It is an excellent exercise to sharpen visual and spatial perception while helping to better appreciate color and environmental differences. It also helps to develop analytical skills, improve mental agility, and entertainingly stimulate visual and working memory.

Organization

As it is a game of moderate difficulty, to develop it correctly, it was necessary to start with the basics and then expand it. This section details the tasks followed to create it:

Basics:

A. The XR device must be represented and operate correctly.
B. The user must be able to teleport into the virtual space.
C. The user must have hands.
D. The user must be able to grab and interact with an object.

Game specific:

E. The user must be able to pick up the portal gun correctly.
F. The gun must be able to fire.
G. The number of bullets should be limited, and there must be a counter.
H. Bullets must be able to make the object they hit disappear.
I. The program must be able to distinguish between objects that are altered and those that aren't.
J. The game must be over if an object that is not an alteration is shot at.
K. The game must be over if all alterations are removed.
L. The game must change scenes properly.
M. The game must create the alters randomly at the start of each game.
N. There must be a timer.

Design:

O. The Game room scene must be fully designed.
P. The Game Over scene with User Interface (UI) must be created.
Q. The Start Menu scene must be created.

Optional:

R. Create a tutorial.
S. Add particles and sounds.
....

Steps

This section details all the steps followed for the game's development.

1. Software setup

1.1. Install the Unity Hub

Before setting up the Unity project, the **Unity Hub's** latest release, 3.2.0, was installed. (https://unity3d.com/get-unity/download)

1.2. Create an account

A new account was created to build a **Unity ID,** given that it was the first time using the development platform. (https://id.unity.com/account/new)

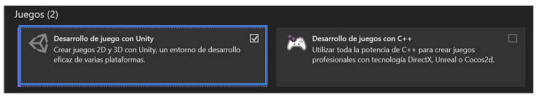
1.3. Install the Unity Editor

After signing into the Unity Hub application, the **Unity Editor 2021.3.6f1 LTS** was installed. This editor was ideal for development as it is stable and will be fully supported for two years.

1.4. Install Microsoft Visual Studio

Microsoft Visual Studio had to be installed on the computer since it is recommended to open and edit scripts. (https://visualstudio.microsoft.com/es/downloads/)

Hence, **Microsoft Visual Studio Community 2022 17.2.6** was installed, which provides new features compared to previous versions. In addition, the **Game Development with Unity** tool was installed, which is helpful for programming in Unity. It could also be installed in Visual Studio by going to Tools > Get Tools and Features... and clicking on Modify.



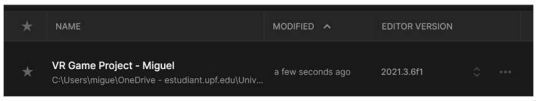
2. VR project setup

2.1. Open a new VR project and set it up

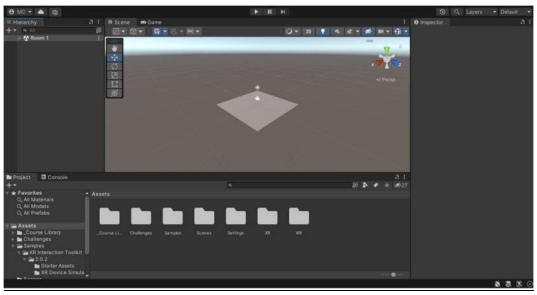
A new Unity project configured for VR was created after opening the Unity application.

The *Create with VR starter project* provided by Unity Learn was downloaded and extracted. This VR Room Project was used as the basis for the project as it contained multiple applicable packages and assets in the public domain.

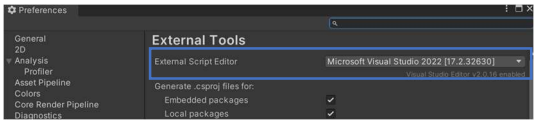
The VR project was renamed, relocated, and added to Unity Hub.



Finally, the project was opened in Unity, and its contents were explored.



Microsoft Visual Studio 2022 was selected as the *External Script Editor* in *Edit > Preferences... > External Tools* to use Visual Studio as an external script editor.

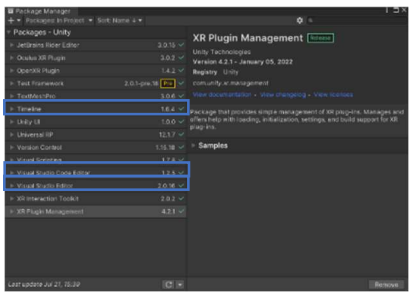


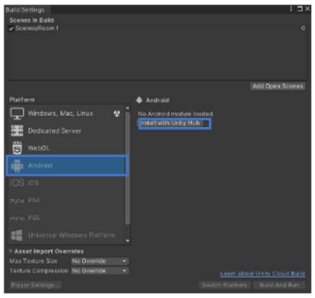
The packages installed in the project (Window > Package Manager) included:

- *XR Plugin Management*
- *XR Interaction Toolkit*
- *Universal RP (Render Pipeline)*

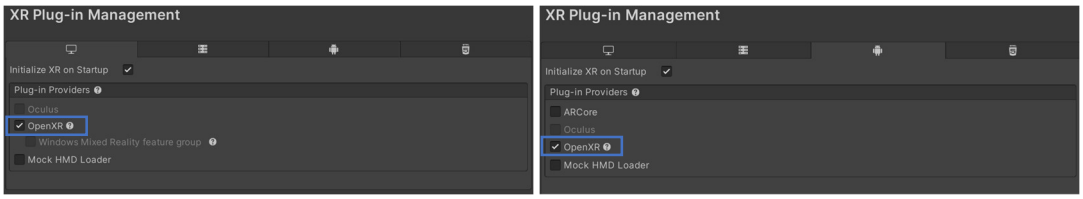
Some other packages, such as the *Oculus XR plugin*, were also installed to provide support for Oculus devices.

The device that would be used was unknown at this point. The *Android* platform was selected in *File > Build Settings*. Then, it was clicked on *Install with Unity Hub* to install and load the Android module.

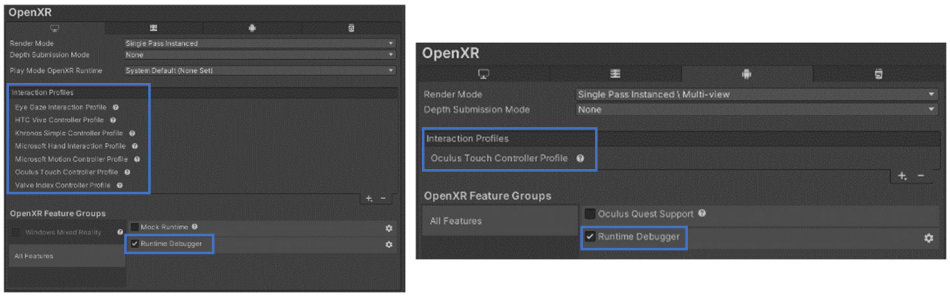




Also, in the *XR Plug-in Management* (*Edit > Project Settings... > XR Plug-in Management*), the *OpenXR plugin* provider was selected in both the Desktop and Android tabs, which allows to seamlessly target a wide range of AR/VR devices.



In the specific *OpenXR* settings, all the *Interaction Profiles* were added to target the maximum number of VR headsets possible. Furthermore, all the *Runtime Debugger OpenXR Feature Groups* were enabled in both Desktop and Android tabs. It's essential to note that allowing the *Mock Runtime* feature would have caused the device simulator configured in section 2.3. to not work.



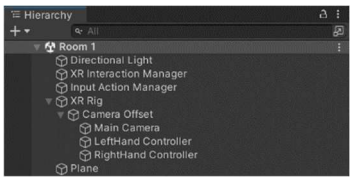
As it can be seen, in the same window, the *Render Mode* could be set.

2.2. Create the Scene

The scene from the project was used as a starting point, which was renamed **Room 1**. It was a simple scene with a grey plane in an open environment. It had other critical components already configured, and the controllers had their functionalities set up.

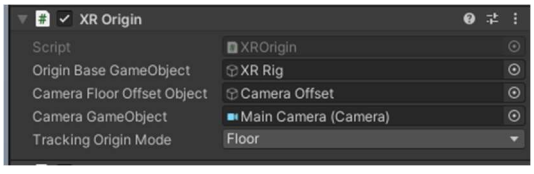
These components, which could be visualized in the *Hierarchy* window, included:

- **Directional light** emits lightning across the entire scene.
- **XR interaction Manager**: the component that manages interactions.
- **Input Action Manager**: the component that manages the input action.
- **XR Rig**: the object that will be moved using locomotion. Thus, it represents the XR device and handles the user's position in the virtual environment.
    - **Main Camera**: the component displays the game to the player tracking its headset position, and is attached directly to the XR Rig. Since the game is not a stationary experience, the Camera *Tracking Type* was set to *Rotation and Position*. Thus, it is expected to work with 6-DoF headsets allowing it to track translational and rotational motion.
    - **Left Hand Controller**
    - **Right Hand Controller**
- **Plane**



It is important to note that the controllers had the *XR Ray Interactor* component, which allowed interaction with objects that were not within the range of the user.

Finally, the *XR Rig Tracking Origin Mode* was set to *Floor* for the camera to adjust correctly to the player's height.



2.3. Run with the Device Simulator

The **Device Simulator** was used to test the application in the editor using the mouse and keyboard.

To do that, the *XR Device Simulator* was added to the scene. Thus, the *XR Device Simulator* Prefab (*Samples > XR Interaction Toolkit > 2.0.2 > XR Device Simulator*) was dragged into the *Hierarchy* window.

There are three devices in VR, and the procedure for using them in the simulator is:

- Step 1: Activate the device to control.
- Step 2: Use the control(s) for that device.

1. Activate devices

Left controller	Headset	Right Controller
Hold Left Shift or toggle with T	Hold Right-click	Hold Spacebar or toggle with Y

2A. Basic controls

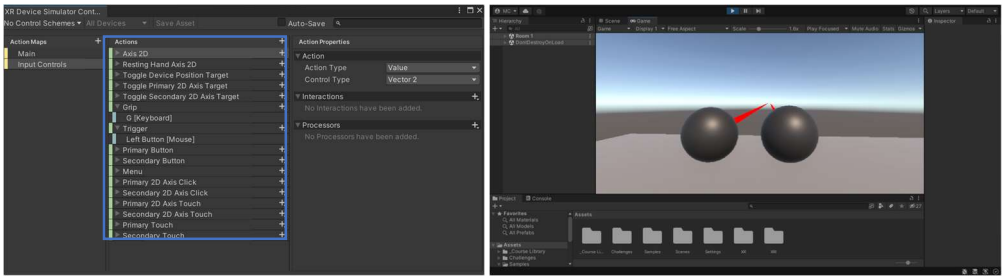
Pan	Move mouse
Rotate	Hold middle mouse
Inverse controls	R
Reset transform	V
Snap turn	A [left] or D [right] (with controller active)

2B. Input controls

Grip	G
Trigger	Left-click
Primary Button	B
Secondary Button	N

source: https://connect-prd-cdn.unity.com/20210604/28db6ca9-aba1-4ac3-a15a-24664daff3ea/Rig%20Simulator%20Keyboard%20Shortcuts.pdf

Many other controls for different actions were available. They could be conveniently viewed by selecting the *XR Device Simulator* in the *Hierarchy* and after, by double-clicking on one of the actions found in the *Inspector* window:



To test the simulator, the *Play* button had to be pressed. It is interesting to note that while simulating the game in the simulator, changes could be made for testing. These changes would not have been applied to the scene.

When running the project on a device, the *XR Device Simulator* must be disabled in the *Inspector* window so that it does not cause problems.

Step A completed.

The XR device must be represented and operating correctly.

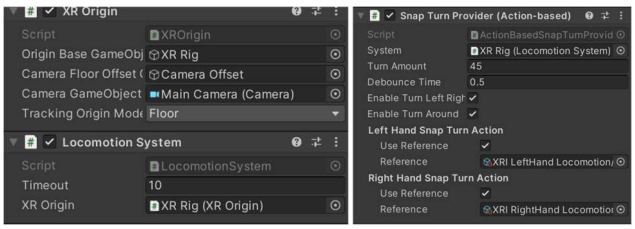
3. VR Locomotion

3.1. Snap turning

It was decided to make it possible for the user to use the joysticks of any handheld controller to rotate a certain number of degrees. This way, the user could turn in place and more easily experience the room without physically moving.

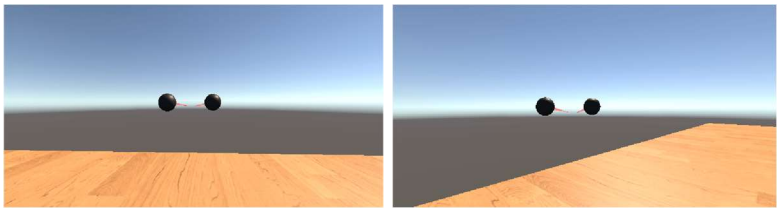
The *XR Rig* object was selected, and the ***Locomotion System*** and ***Snap Turn Provider (Action-based)*** components were added.

The *XR Origin* component from the Inspector was dragged to the *XR Origin* property slot inside the *Locomotion system* component. Moreover, the *Locomotion System* component was dragged to the *System* property slot inside the *Snap Turn Provider* component.



With this setup, the user could snap turn 45°, moving the joysticks to the sides (*Enable Turn Left Right* property), and 180° when pressed down (*Enable Turn Around* property).

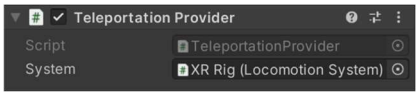
Using the XR device simulator, the movement could be simulated by pressing *T* or *Y* to toggle one of the controllers and then *A* or *D* to simulate the 45° snap turn to the left and the right, respectively. Using *S*, the 180° snap turn could be simulated.



3.2. Teleportation

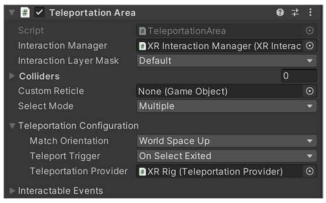
Teleportation allows the user to move around the environment with low sickness so that the play area is not limited.

To enable teleportation, the *XR Rig* object was selected, and the ***Teleportation Provider*** component was added. Furthermore, the *Locomotion System* component was dragged to the *System* property slot inside the component.

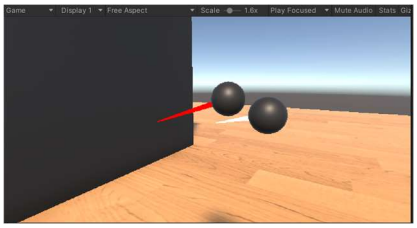


3.3. Teleportation area

The next step was to make a teleportation area or anchor. The ***Teleportation Area*** component was added to the plane to convert it to a teleportation area. Furthermore, the *XR Rig* object was assigned to the *Teleportation Provider* property, which could be found at the bottom of the *Teleportation Area* component.



When pointing anywhere on the plane, the line renderer turned white, and the player could teleport using the grip button on the controller. Moreover, the user was not able to teleport through physical objects.



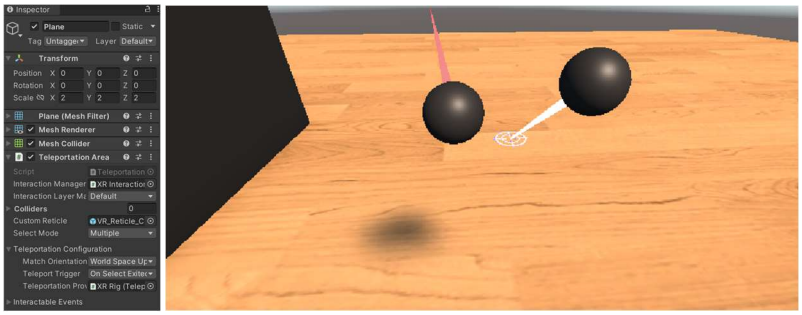
Using the XR device simulator, the controller's rotation could be simulated by toggling a controller and rotating it by holding the middle mouse button. Pressing the *G* button (grip), the user could teleport.

Step B completed.

The user must be able to teleport in the virtual space.

3.4. Teleportation ray customization

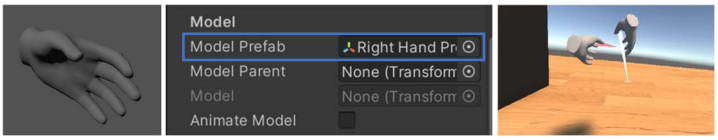
To conclude, the ray was customized. New events were added, the beam color was changed, and a reticle was added. This reticle was shown whenever the user pointed toward the plane to provide more visual feedback.



4. Object interaction

4.1. Hand presence

Rather than using spheres, a model to represent the user's hands in VR was used. A 3D model of a hand was downloaded. Specifically, the Oculus hand models were downloaded and dragged into the controllers *Model Prefab* attribute.

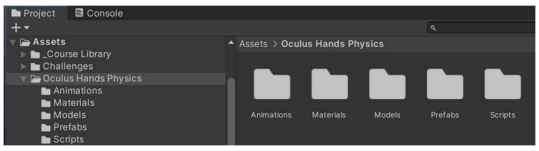


Step C completed.

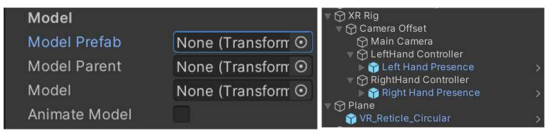
The user must have hands.

4.2. Hand animation

At this stage, the hands were immobile when performing actions such as gripping. The *Oculus Hand Physics* package provided by *Valem Tutorials* (https://www.youtube.com/c/ValemTutorials) was used to add animations to the hands.



The package included the same hand models used and their animations configured. The hand models from the controllers were removed, and the *Hand Presence* prefabricated were dragged into the hand controllers so that the hands followed the controllers and produced the corresponding animations.



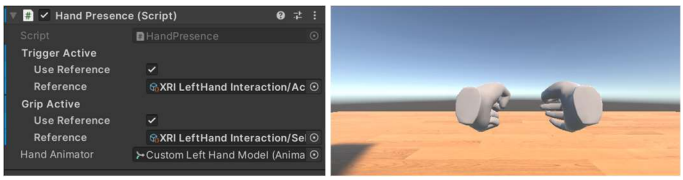
However, the animations were not working in the device simulator since the *HandPresence* script, which managed them, detected the actions according to the input device. The simulator was not seen as an input device. Hence, the *HandPresence* script was modified entirely to work more efficiently.

The input actions provided in the *XR Interaction Toolkit* were used. The references provided in this package allowed the detection of the input actions and their values. The script was programmed to set the trigger input action (activate) and grip input action (select) values into the animator values in every frame, so the animation occurs.

It is important to notice that the `[SerializeField] private` made a variable private but accessible from the *Inspector*.

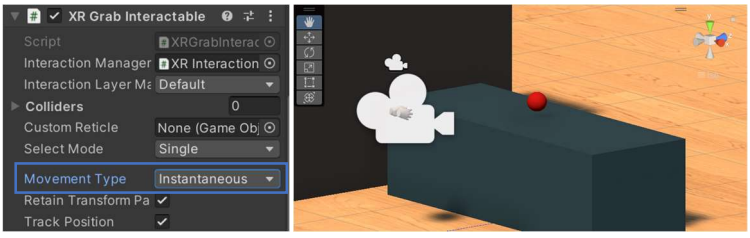
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR;
using UnityEngine.InputSystem; //to listen to an input

public class HandPresence : MonoBehaviour
{
    [SerializeField] private InputActionProperty TriggerActive; //to get the trigger action
    [SerializeField] private InputActionProperty GripActive; //to get the grip action
    [SerializeField] private Animator handAnimator; //get the hand animator

    // Start is called before the first frame update
    void Start()
    {
        //to read the inputs correctly they must be enabled
        TriggerActive.action.Enable();
        GripActive.action.Enable();
    }

    void Update()
    {
        float triggerValue = TriggerActive.action.ReadValue<float>(); //read the value of the trigger action
        handAnimator.SetFloat("Trigger", triggerValue); //set the trigger value in the animator to the input action value

        float gripValue = GripActive.action.ReadValue<float>(); //read the value of the grip action
        handAnimator.SetFloat("Grip", gripValue); //set the grip value in the animator to the input action value
    }
}
```

Finally, all the fields of the script component were completed. The script could be found in both *Hand Presence* objects. The animator attribute was already configured. For the *Trigger Active* feature, the *Activate Value* input actions references of the corresponding hand were selected. The same was performed for the Grip Active attribute; the *Select Value* input actions references of the corresponding hand were selected.

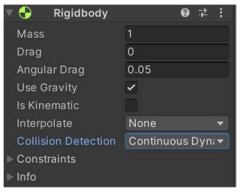


4.3. Grabbable object

An object was added to the scene. Specifically, a sphere was created in the background, and the *XR Grab Interactable* component was added to it, making it grabbable. In this component, the throwing and gripping characteristics of the object were configured. For instance, the *Movement Type* of the object could be set: *Kinematic*, *Velocity Tracking*, or *Instantaneous*. The ***Instantaneous*** movement type was selected so that the object got instantaneously into the hand when grasped and behaved appropriately when colliding with other entities.

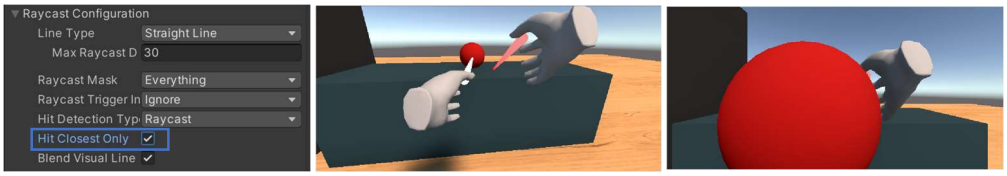


The *XR Grab Interactable* component automatically added a ***Rigidbody*** part so that the object was affected by physics. The *Collision Detection* was set to *Continuous Dynamic* to prevent the object from going through the floor when dropped.



4.4. Distance grab

Without the need to configure anything, the beam allowed the user to grasp an object from a distance. Thus, the ray could be used for both teleporting and grabbing. The beam was configured to *Hit Closest Only* so that the glow did not continue after hitting an object and therefore did not pick up two objects at the same time.



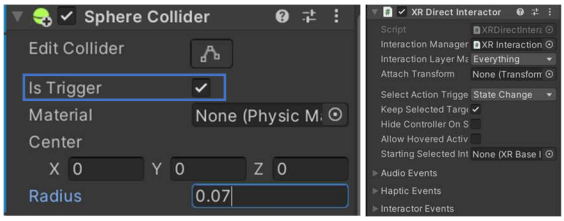
Using the XR device simulator, the object could be grabbed and held by pointing the beam at the object and pressing and holding the *G* key.

4.5. Grab interaction

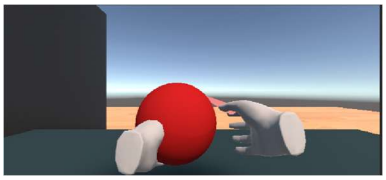
At this point, the user could grasp objects at a distance but could not get something touched.

To this end, the ***XR Direct Interactor*** and ***Sphere Collider*** components were added to the controllers. The *Is Trigger* property of the *Sphere Collider* component was enabled to allow grabbing an object. The radius of the *Sphere Collider* was adjusted. This value sets the contact zone radius for holding an object.

The controllers did not allow the addition of the *XR Direct Interactor* since an object could only contain one *XRBaseInteractor* component, and the *XR Ray Interactor* component was already present. Thus, both components were added to the *Hand Presence* objects.

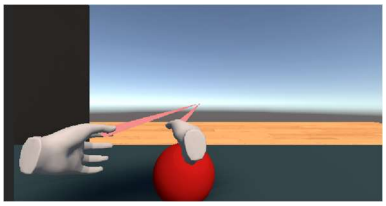


The object needed the ***XR Grab Interactable***, ***Rigidbody***, and ***Sphere collider*** components to be grabbable. The sphere created already had them.

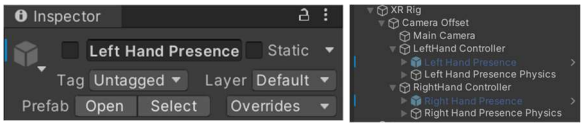


4.6. Hands physics

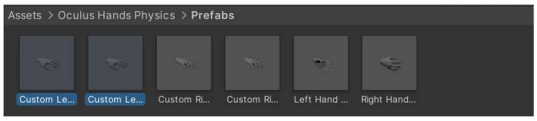
At this point, the user could pick up objects, but his hands could not interact with them, i.e., the hands did not collide with objects.



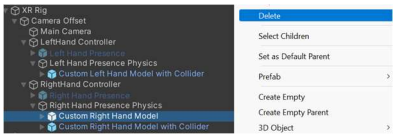
To solve this, physical capabilities were given to the hand. The *Left-* and *Right-Hand Presence* objects were disabled, duplicated (*CTRL+D*), and renamed.



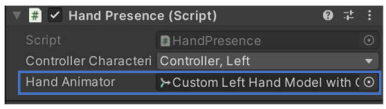
Then, a *Hand Model with a Collider* was dragged to the *Hand Presence Physics*. This model could be found in *Oculus Hands Physics > Prefabs* and added a ***Capsule Collider*** component to each of the bones of the hand.



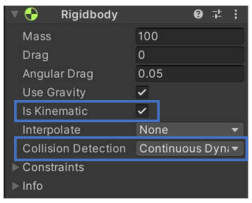
Moreover, the *Custom Hand Model* inside both *Hand Presence Physics* was deleted.



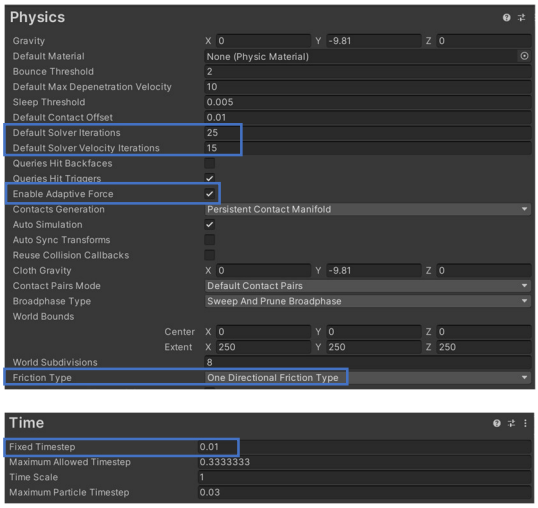
In addition, the hand models with colliders were dragged to the *Hand Animator* of the corresponding *Hand Presence Physics*.



Finally, a rigid body was added to the hand physics so that they could interact with objects. The ***Rigidbody*** component was added to *Hand Presence Physics* with the *Is kinematic* property selected. Furthermore, the *Collision Detection* was changed to *Continuous Dynamic* so that the hands did not pass through any rigid body object.



At this point, the user could interact physically with the sphere using their hands. However, the interaction was not precise. Hence, some physics settings were modified (*Edit > Project Settings... > Physics*). The ***Default Solver Iterations*** and ***Velocity Iterations*** were incremented to 25 and 15, respectively. These settings could affect the game's performance. Moreover, the ***Adaptive Force*** was enabled, the ***Friction Type*** was set to *One Directional Friction Type,* and the ***Solver Type*** to *Temporal Gauss-Seidel*. To conclude, inside the *Time* setting, the ***Fixed Timestep*** was reduced to 0.01.



Step D completed.

The user must be able to grab and interact with an object.

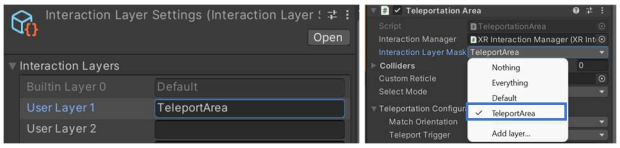
5. Ray interactors

At this stage, a single ray allowed the player to pick up objects and teleport.

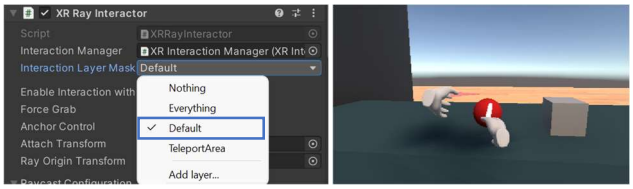
Usually, in VR games, this is not the case. There is one beam for each action. In addition, teleporting does not use the trigger button.

Hence, some changes were made so that the teleport beam was activated by pressing the left joystick upwards, and by pressing the trigger or grip button, the user would teleport. Moreover, the teleport ray was curved to facilitate teleportation and made to deactivate when an object was grabbed.

The scene ray had to perform the sole function of grabbing objects. Thus, a new interaction layer that would allow only interaction with teleport areas was created. This new layer was called *TeleportArea*. To add it, it was clicked on *Add Layer...* after clicking on the *Interaction Layer Mask* drop-down inside the *Teleportation Area* component of the plane object. Then, the *Interaction Layer Mask* of the *Teleportation Area* component was changed to only this new layer.

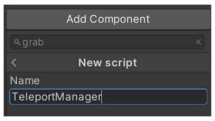


Afterward, in the *XR Ray Interactor* of both controllers, the *TeleportArea* layer was disabled in the *Interaction Layer Mask* so that both rays do not allow teleportation.



For the ray of the left controller to become the teleport beam via a script, the *Unity Adventure tutorial: "Teletransportación en Realidad Virtual | Unity - XR Interaction Toolkit"* was followed.

A new component in the *XR Rig* object called *TeleportManager* was created to control the teleport system.

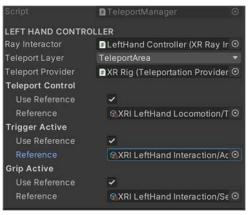


The script was programmed to perform the desired functions. By pressing the activation teleport button (joystick up), the teleport ray was activated or deactivated accordingly. In addition, pressing the trigger or grip button teleported the user when the teleport ray was active.

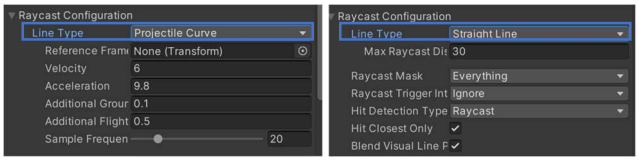
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;
using UnityEngine.XR.Interaction.Toolkit;

public class TeleportManager : MonoBehaviour
{
    [SerializeField] private XRRayInteractor rayInteractor; //to store the ray Interactor component, the object that contains it must be dragged

    [SerializeField] private InteractionLayerMask TeleportLayer; //to get the interaction layer mask
    [SerializeField] private TeleportationProvider TeleportProvider; //to get the teleportation provider

    [SerializeField] private InputActionProperty TeleportControl; //to get the action I want to control the teleportation
    [SerializeField] private InputActionProperty TriggerActive; //to get the trigger action
    [SerializeField] private InputActionProperty GripActive; //to get the grip action

    private bool isTeleportActive = false; //to know if the teleport is active, initially not active
    private InteractionLayerMask InitialInteractionLayer; //to store the initial interaction layers
    private List<IXRInteractable> interactables = new List<IXRInteractable>(); //to store the list of valid ray targets

    // Start is called before the first frame update
    void Start()
    {
        //to read the inputs correctly they must be enabled
        TeleportControl.action.Enable();
        TriggerActive.action.Enable();
        GripActive.action.Enable();

        //now I need to know when the teleport is activated or not (the set action is performed)
        TeleportControl.action.performed += OnTeleportControl;

        InitialInteractionLayer = rayInteractor.interactionLayers; //store initial interaction layers
    }

    // Update is called once per frame
```

```
    void Update()
    {
        //verify the teleport is active
        if (!isTeleportActive)
        {
            return; //do nothing
        }
        //if the teleport is active and the trigger button is not pressed
        if (!TriggerActive.action.triggered)
        {
            return;
        }
        //now the teleport is active and the trigger button is pressed
        //it must be a valid zone to teleport
        rayInteractor.GetValidTargets(interactables);
        //if no valid teleport area
        if (interactables.Count == 0)
        {
            return;
        }
        //get the teleportation point
        rayInteractor.TryGetCurrent3DRaycastHit(out RaycastHit hit);
        TeleportRequest request = new TeleportRequest();
        if (interactables[0].interactionLayers == 2) //teleportation area layer mask
        {
            request.destinationPosition = hit.point;
        }
        //teleport to the point
        TeleportProvider.QueueTeleportRequest(request);
    }

    private void OnTeleportControl(InputAction.CallbackContext Obj)
    {
        if (isTeleportActive == true) //if teleport is active, deactivate it
        {
            turnOffTeleportation();
        }
        else
        {
            //verify that the grip is not pressed
            if (GripActive.action.phase != InputActionPhase.Performed)
            {
                turnOnTeleportation();
            }
        }
    }

    private void turnOnTeleportation()
    {
        //activate the teleportation
        isTeleportActive = true;

        //change the line type of the ray to curve when teleport is active
        rayInteractor.lineType = XRRayInteractor.LineType.ProjectileCurve;
        //to allow the teleport ray to have as interactor layer mask the teleport layer
        rayInteractor.interactionLayers = TeleportLayer;
    }

    private void turnOffTeleportation()
    {
        //deactivate the teleport
        isTeleportActive = false;

        //change the line type of the ray back to straight when teleport is active
        rayInteractor.lineType = XRRayInteractor.LineType.StraightLine;
        //to reset the interactor layer mask
        rayInteractor.interactionLayers = InitialInteractionLayer;
    }
}
```

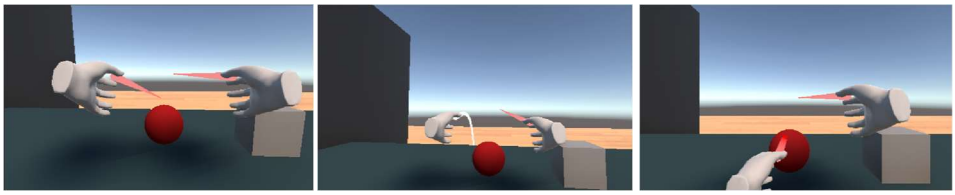
Finally, all the fields of the component were completed. The *Ray Interactor* was the *Left-Hand Controller* where the *XR Ray Interactor* to be substituted was found. The *Teleport Layer* was the teleportation interaction layer mask. The *Teleport Control* was the *LeftHand Locomotion/Teleport Mode Activate* action, i.e., the button that activates the teleport mode. The *Trigger* and *Grip Active* were the *hand's Activate and Select input actions*, respectively.



The teleport beam configuration could be done by momentarily changing the beam's *Line Type* to *Projectile Curve*. To work correctly, the Line Type had to be set back to a Straight Line.



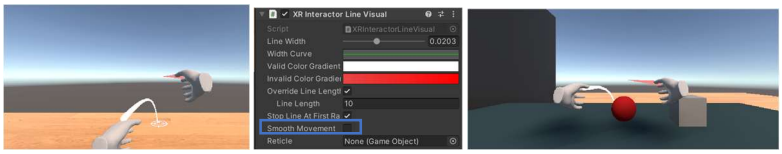
In the simulator, pressing the key *W* enabled or disabled the teleportation ray. The rig would teleport when pushing the trigger control (mouse left-click).



6. Issues

6.1. Teleport ray issue

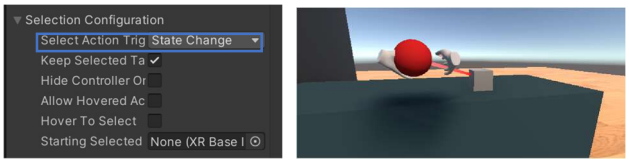
The teleportation beam did not follow the controller's movements correctly. The beam moved into position with some delay. To solve this, I disabled the *Smooth Movement* property in the *XR Interactor Line Visual* of the controllers.



6.2. Grabbing ray issue

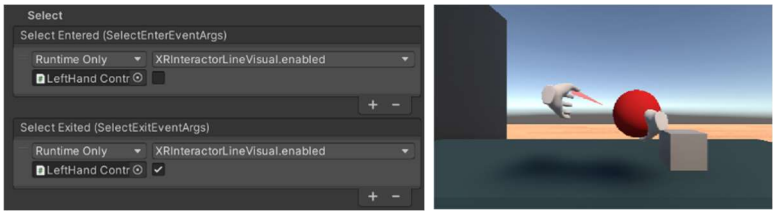
The ray was still active when an object was grabbed. Therefore, if the beam pointed to another entity, it was grabbed even if an object was held.

To fix that, the *XR Ray Interactor Select Action Trigger* was set to ***State Change*** in both controllers.

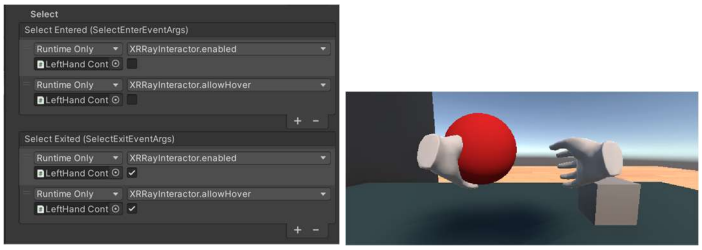


The problem was partially solved; the lightning was still visible.

Hence, the *XR Interactor Line Visual* was disabled when holding an object. To do that, *Interactor Events* were added to both controllers' *XR Ray Interactor* components. An event to disable the *XR Ray Interactor Line Visual* on *Select Entered* and another one to enable it on *Select Exited* were added. The corresponding controller was added to the object variable, and the *XRInteractorLineVisual > bool enabled* was selected in the function list. To allow the line to be visible, the checkbox was selected.

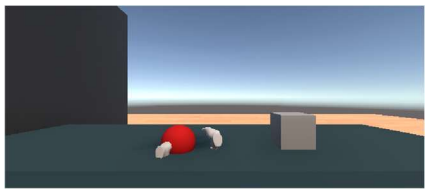


The issue was not solved when an object was grabbed using the *Direct Interactor*. Thus, *some Interactor Events* were added to the *XR Direct Interactor* component in both hand physics. The XR Ray Interactor and its hover property were disabled on *Select Entered*. On *Select Exited*, they were enabled again. The corresponding controller was dragged to the object variable, and the *XRRayInteractor > bool enabled* and *XRRayInteractor > bool allowHover* were selected in the function list.



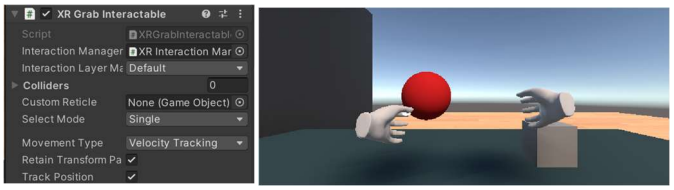
6.3. Non-rigid bodies

The hands and grabbed objects were passing through objects that did not have a rigid body.

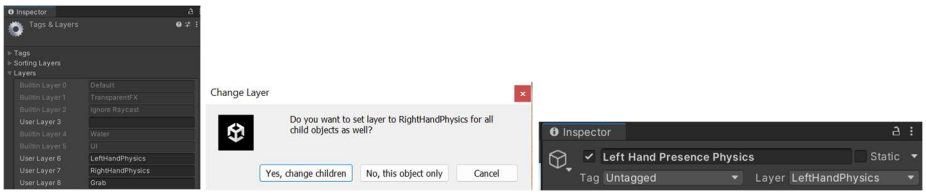


6.3.1. Grabbed objects and non-rigid bodies

Solving the problem related to grabbing objects could have been manageable by changing the motion type of the *Grab Interactable* component of the object to ***Velocity tracking***. This movement type made the ball collide with any other collider of the scene while grabbed, even if it did not have a rigid body. However, some jittering occurred when the object was moved quickly when held in hand. The real problem was that because the hands had physical properties, the object collided with them, and the grip looked unnatural.

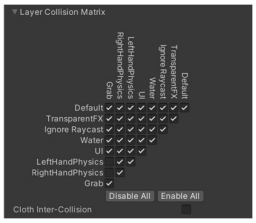


To solve this, different layers were created. A layer was made for each hand physics (*LeftHandPhysics* and *RightHandPhysics*) and the grasped objects (*Grab*). The hand physics layers were assigned to the corresponding object and all children.

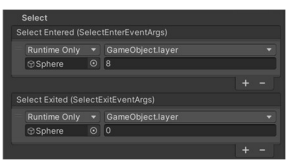


The idea was that the *Grab* layer would not collide with the layers of the hands. Thus, this layer was assigned to the objects when they were grabbed. Once released, the layer was set back to the default interaction layer.

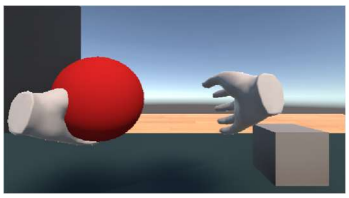
Hence, the inner collision between the *Hand Physics Layers* and the *Grab Layer* was disabled in the *Layer Collision Matrix* by going to *Edit > Project Settings... > Physics*.



Finally, an event in every grabbable object of the scene was added. Specifically, in the *XR Grab Interactor* component, an event was added for the *Select Entered* (i.e., when the object is grabbed) and *Select Exited* (i.e., when the object is released). The *GameObject > layer* was selected as the function to configure the object layer. When grabbed, the *Grab* layer (8) was set. When released, it was returned to the *Default* layer (0).



Thus, the issue with the grabbed objects was fixed; objects did not collide with the hands anymore.



The events needed to be added to all the grabbable objects to work properly with the physic hands.

6.3.2. Physic hands and non-rigid bodies

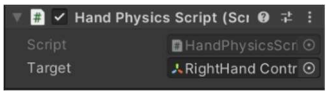
The *Valem Tutorials: How to Make Physics Hands in VR - PART 2* video was followed to fix the hand issue.

First, the *Is Kinematic* property was disabled for both *Hand Presence Physics*. This property caused the hands not to collide with non-rigid bodies.

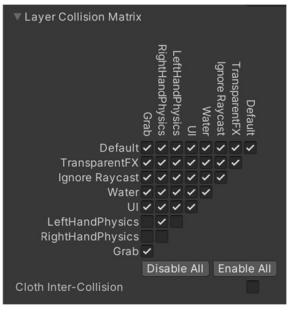
However, disabling it made the hands not follow the controller's position correctly. Thus, a script was created for both *Hand Presence Physics*. This ***HandPhysicsScript*** was programmed following the tutorial to make the hands follow the position and rotation of the controller using their rigid body.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class HandPhysicsScript : MonoBehaviour
{
    [SerializeField] private Transform target;
    private Rigidbody rb;

    // Start is called before the first frame update
    void Start()
    {
        //access the rigid body component at the start of the game
        rb = GetComponent<Rigidbody>();

    }

    void FixedUpdate()
    {
        //make the rigid body follow the pos of the target
        //change the vel of the object to go to the target pos
        rb.velocity = (target.position - transform.position) / Time.fixedDeltaTime;
        //make the rigid body follow the rotation of the target
        Quaternion rotationDifference = target.rotation * Quaternion.Inverse(transform.rotation);
        rotationDifference.ToAngleAxis(out float angleInDegree, out Vector3 rotationAxis);

        Vector3 rotationDifferenceInDegree = angleInDegree * rotationAxis;

        //compute angular velocity in radiants
        rb.angularVelocity = (rotationDifferenceInDegree * Mathf.Deg2Rad / Time.fixedDeltaTime);
    }
}
```

Finally, the corresponding *Hand Controllers* were dragged to the target variable in the *HandPhysicsScript* component.

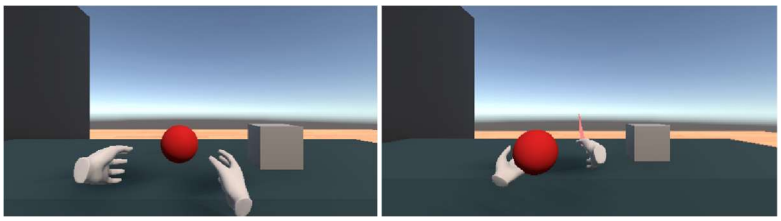


The hands were working correctly, but there was a jittering issue because of the hands colliding with each other. To solve this, the inner collision between the same hands was disabled in the *Layer Collision Matrix*.



It is important to note that in the end, it was decided to disable the interaction between different hands just in case it could cause any problem when playing.

With all this, the problem was solved. Hands and objects were colliding with non-rigid bodies.



6.4. Object release

The solution in the previous section led to another problem. When the object was released, it collided with the physical hands in the first frames. This caused the object to shoot out.

To address this problem, the idea was to add some delay in resetting the default object layer.

Thus, the function used in the *Select Exited* event was implemented in a script with a delay.

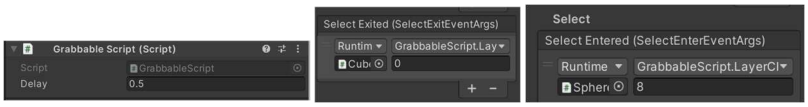
A new *GrabbableScript* was created and added as a component of the grabbable objects. Inside the script, a method (*LayerChange()*) was invoked after an amount of time. The delay was a variable since the Unity inspector could only assign callback functions that take zero or one argument. Moreover, the layer number was stored in a private variable since the *Invoke* method did not allow the passing of any parameter to the method invoked. The invoked method applied the layer change to the object and children.

In addition, a method to apply the *Select Entered* layer change for the object and children was created.

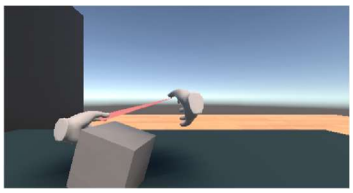
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GrabbableScript : MonoBehaviour
{
    private int layernum; //store the layer number
    [SerializeField] private float delay = 0.5f;

    public void LayerChangeWithDelay(int layer)
    {
            layernum = layer;
            Invoke("LayerChange", delay); //invoke the change layer method in delay sec
    }

    public void LayerChange()
    {
        //for the object and childs
        Transform[] childs = GetComponentsInChildren<Transform>(); //list containing the object and its childs
        foreach (Transform child in childs)
        {
            try
            {
                child.gameObject.layer = layernum;
            }
            catch
            {
                continue;
            }

        }
    }


    public void LayerChangeGrab(int layer)
    {
        //for the object and childs
        Transform[] childs = GetComponentsInChildren<Transform>(); //list containing the object and its childs
        foreach (Transform child in childs)
        {
            try
            {
                child.gameObject.layer = layer;
            }
            catch
```

```
            {
                continue;
            }

        }
    }
}
```

To conclude, the *Select Entered* and *Exited* events were deleted. A new event with the corresponding *GrabbableScript* method as the function was added. For the *Select Exited*, the *LayerChangeWithDelay* method was called, setting the layer to the *Default* (0). Furthermore, a delay value of 0.5 sec was set. This value was obtained by trial and error. For the *Select Entered*, the *LayerChangeGrab()* method was called with the Grab layer (8) as the parameter.



Hence, the program behavior was appropriate when releasing an object.



The events and script needed to be added to all the grabbable objects to work properly.

6.5. Grabbed object

When an object was held with one hand and grasped using the other hand's beam, a collision occurred between the physical hand and the object. This was because of the delay introduced. If the object was held before the delay expired, it was first switched to the layer in which they do not collide (*Select Entered*). When the delay passed, it was swapped to the layer in which they differed (*Select Exited*) even if the object was still grabbed.

To solve this, a variable was added to indicate whether the object was grabbed. This variable became true when the select event happened and false when the release event occurred. Thus, the following changes were made to the *GrabbableScript*.

Variable added:

```
    private bool isGrabbed = false;
```

Methods modified and added:

```
    public void LayerChange()
    {
        if (!isGrabbed)
        {
            //for the object and childs
            Transform[] childs = GetComponentsInChildren<Transform>(); //list containing the object and its childs
            foreach (Transform child in childs)
            {
                try
                {
                    child.gameObject.layer = layer;
                }
                catch
                {
                    continue;
                }
            }
        }
    }

    public void objectGrabbed()
    {
        isGrabbed = true;
    }

    public void objectReleased()
    {
        isGrabbed = false;
    }
```

The call to the *objectGrabbed* function was added to the *Select Entered* event. The call to the *objectReleased* method was added to the *Select Exited* event.

6.6. Velocity tracking jittering

Due to the solution implemented in section 6.3.1. jittering occurred on the grabbed object when the user controllers were moved quickly. This happened because velocity tracking used the physics engine to move the object (via velocity), but the standard movement systems moved the player via transforms.

A straightforward way to alleviate this was to make the object a child of the *XR rig* when selected in the *GrabbableScript*.

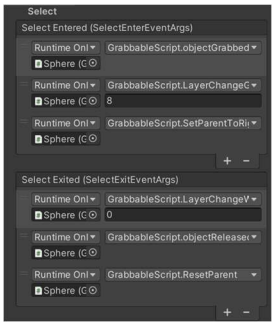
Variable added:

```
    private Transform parent;
```

Methods added:

```
    void Start()
    {
        parent = transform.parent; //store the parent
    }

    public void SetParentToRig()
    {
        transform.SetParent(GameObject.Find("XR Rig").transform);//get the rig
    }
    public void ResetParent()
    {
        transform.SetParent(parent);
    }
```

The implemented methods were called on the *Select Entered* and *Exited* events accordingly.



Needed to be added to all the grabbable objects to work properly.

7. Customization

7.1. Invisible hands

In case of future problems with hands when grasping an object, it was implemented that the hands could be invisible.

The problem was that the interactors' functions to hide the controller when grabbing an object (*Hide Controller on Select*) did not work because of how the hand model was introduced. Therefore, programming was needed. Thus, within *HandPhysicsScript*, a variable was created to contain the model object and a function to disable it when desired.

Variable added:

```
    [SerializeField] private GameObject targetModel;
```

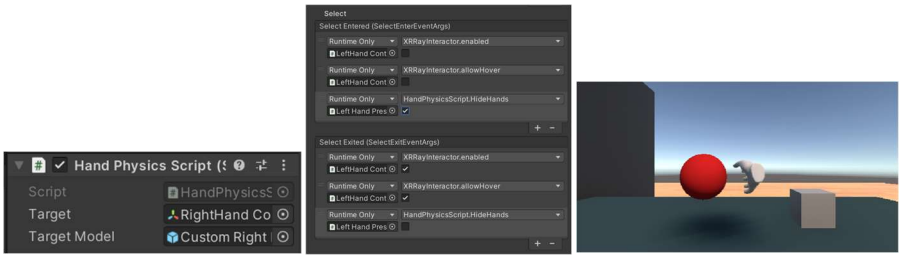
Method added:

```
    public void HideHands(bool hide) //to  hide or not the hands
    {
        targetModel.SetActive(!hide);
    }
```

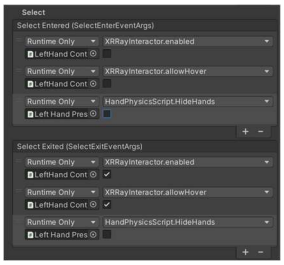
It had to be called when interacting with an object to work correctly. Thus, the corresponding *Hand Model with Collider* was the target model.

Then, in the *XR Direct Interactor* of the controllers (*< Hand Presence Physics*) and the *XR Ray Interactor (< Hand Controller*), an event for the *Select Entered* and *Exited* events was added. The corresponding

Hand Presence Physics (which contained the script) was dragged as the object. Finally, the *HideHands()* function was enabled for the *Select Entered* and disabled for the *Select Exited*.



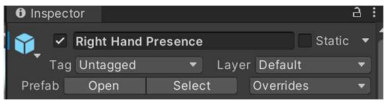
Thus, the hands were able to disappear when grabbing an object. However, it was decided not to make the hands invisible for the time being.



7.2. Non-physical hands

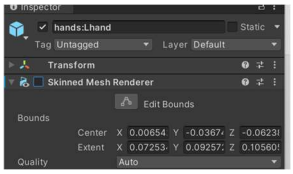
It was decided to show the user's hands when they move a certain distance away from the physical hands with colliders.

To this end, the *Left-Hand* and *Right-Hand Presence* objects, disabled in section 4.6, were re-enabled. These were simply objects that contained the configured animations and the hand model without colliders.



The *Sphere Collider* and *XR Direct Interactor* components were deleted because these hands should not interact with any object.

The *Skinned Mesh Renderer* component was used. This component was disabled. Then, when the position of the controller was at a certain distance from the physical hands, the component was re-enabled so that these non-physical hands were displayed. The object with this component was found in *Hand Presence < Custom Hand Model < hands:hands_geom < handsLhand/handsRhand* for both *Hand Presence* objects.



Some code was added to the *HandPhysicsScript* to enable them when the threshold value was exceeded. The threshold value obtained by trial and error could be configured from the Unity inspector.

Variables added:

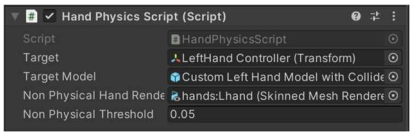
```
    [SerializeField] private Renderer nonPhysicalHandRenderer; //to store the renderer and enable it and disable it
    [SerializeField] private float nonPhysicalThreshold = 0.05f; //threshold to show the non physical hand
```

Method added:

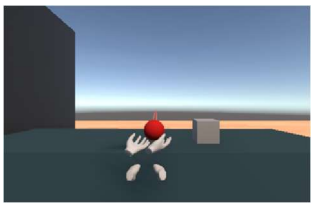
```
    void Update()
    {
        //get distance between controller and physical hands
        float distance = Vector3.Distance(transform.position, target.position);

        //if distance bigger than threshold then enable (show) the non physical hands
        //else disable them
        if (distance > nonPhysicalThreshold)
        {
            nonPhysicalHandRenderer.enabled = true;
```

```
        }
        else
        {
            nonPhysicalHandRenderer.enabled = false;
        }

    }
```

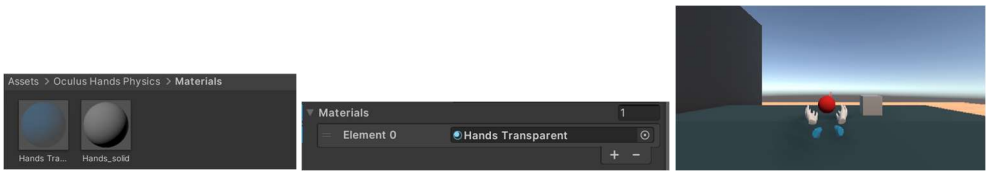
Then, the corresponding hand object (*handsLhand/handsRhand*) was assigned to the *Non-Physical Hand Renderer* attribute inside the *HandPhysicsScript* component (< *Hand Presence Physics objects*).



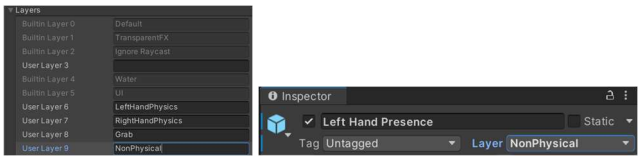
At this point, the non-physic hands were displayed. However, they were of the same material as the physic hands. Moreover, they passed through objects that did not have a rigid body. Hence, they were not well visualized, and the rays were still operative but in the position of the non-physical hands.



Thus, the *Skinned Mesh Renderer* component's material was changed to transparent material from the downloaded *Oculus Hands Physics* folder.

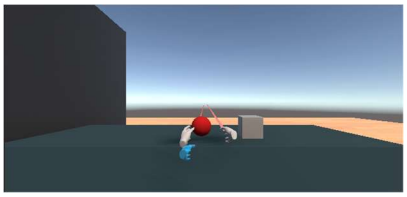


To make them visible in front of any object, a new layer for the non-physical hands was created. Moreover, it was assigned to both *Hand Presence* objects representing the non-physical hands and for all children.



Then, the *Universal Render Pipeline Asset_Renderer* asset was used. It could be found by searching in the *Project* window. A *Renderer Feature* was added—specifically, the *Render Objects (experimental)* feature. *Transparent* was selected as the *Queue* and the *NonPhysical* layer as the *Layer Mask* in the *Filters* drop-down. The *Depth* in the *Override* drop-down was enabled. To conclude, the *Depth Test* was set to *Greater* so that it always displayed the hands in front of any other entity.

At this point, the non-physical hands were shown in front of any other object.



To solve the ray issue, the *Ray Interactor Visual,* when the non-physical hand was displayed, had to be disabled. In addition, the *Ray Interactor* had to be deactivated when the hand was not grabbing an object. The interactor would not be halted entirely because a grasped object would be released. Furthermore, it was not enough to disable the visual because the user could also teleport. Thus, some changes were applied to the *HandPhysicsScript*.

Namespace added:

```
using UnityEngine.XR.Interaction.Toolkit;
```

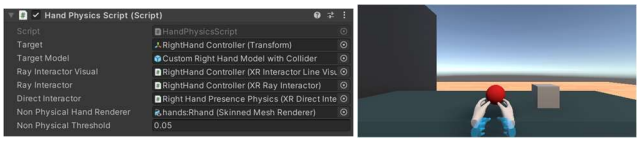
Variables added:

```
    [SerializeField] private XRInteractorLineVisual rayInteractorVisual; //to get the ray interactor visual component
    [SerializeField] private XRRayInteractor rayInteractor; //to get the ray interactor component

    [SerializeField] private XRDirectInteractor directInteractor; //to get the direct interactor component
```

Method modified:

```
    void Update()
    {
        //get distance between controller and physical hands
        float distance = Vector3.Distance(transform.position, target.position);

        //if distance bigger than threshold then enable (show) the non physical hands and disable the ray
        //else disable the hands and enable the ray
        if (distance > nonPhysicalThreshold)
        {
            nonPhysicalHandRenderer.enabled = true;
            if (rayInteractor.isSelectActive) //if its holding an object
            {
                rayInteractorVisual.enabled = false; //only make it non visual
            }
            else
            {
                rayInteractor.enabled = false;//if not disable it
            }
        }
        else
        {
            nonPhysicalHandRenderer.enabled = false;
            rayInteractor.enabled = true;
            if (rayInteractor.isSelectActive || directInteractor.isSelectActive) //if its holding an object
            {
                rayInteractorVisual.enabled = false;
            }
            else
            {
                rayInteractorVisual.enabled = true;
            }
        }

    }
```

Finally, the objects (which had the interactor components) were dragged to the script attributes.



7.3. Ray hit an object

Grabbable objects were expected to turn white when the user was trying to grasp them.

To do this, an action for the *Hover Entered* and *Hover Exited* event was added in the *XR Grab Interactable component*. When the *Hover Entered* event occurred (valid ray hitting the object), the material was changed to white. When *Hover Exited*, it was changed back to the usual material.

For both events, a method in *GrabbableScript* was created. The *Hover Entered* method changed the material for the object and all children if the object was not grabbed. It also stored the material of every child in a list so that it could be reset after. For the *Hover Exited*, the method reset the initial material of the object and children according to the material stored.

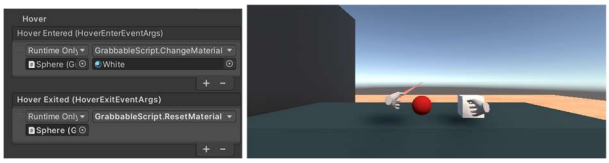
Variable added:

```
    private List<Material> childsIniMat = new List<Material>(); //store the object material and its childs
```

Methods added:

```
    public void ChangeMaterial(Material material)
    {
        if (!isGrabbed)
        {
```

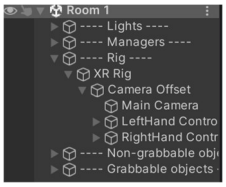
```
            Transform[] childs = GetComponentsInChildren<Transform>(); //list containing the object and its childs
            foreach (Transform child in childs)
            {
                try
                {
                    if (child.GetComponent<Renderer>() != null)
                    {

                        childsIniMat.Add(child.GetComponent<Renderer>().material);//store the material
                        child.GetComponent<Renderer>().material = material;//change the material
                    }
                }
                catch
                {
                    continue;
                }

            }
        }
    }

    public void ResetMaterial()
    {
        if (childsIniMat.Count > 0)
        {
            Transform[] childs = GetComponentsInChildren<Transform>();
            int i = 0;
            foreach (Transform child in childs)
            {
                try
                {
                    if (child.GetComponent<Renderer>() != null)
                    {
                        child.GetComponent<Renderer>().material = childsIniMat[i];//reset its material
                        i++;
                    }
                }
                catch
                {
                    continue;
                }

            }
        }
    }
```



Needed to be added to all the grabbable objects to work properly.

7.4. Organized Hierarchy

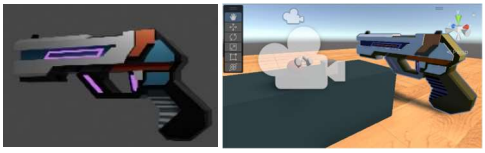
Before getting too many objects in the Hierarchy, an organizational system based on categories was developed.



8. Portal gun

8.1. Design

The portal gun used was a handgun-free asset from the Unity Asset store (https://assetstore.unity.com/packages/3d/props/guns/sci-fi-handgun-225160) from the Life Hacker designer:



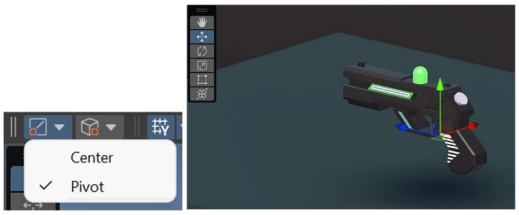
The pistol was downloaded, imported, and added to the project. In addition, its colors were changed, and some elements were added. Finally, the gun was configured, adjusting its size and adding all the necessary components.

8.2. Attach point

A gun is an object that must be grabbed at a very particular point and orientation. Thus, an attached point was assigned to the pistol.

First, the *Toggle Tool Handle Position* was changed to *Pivot* and the *Rotation* to *Local* so that the blue axis represented the forward axis of the controller and the green axis represented the top axis of the controller.

In the *Hierarchy*, an empty child object named *GunAttach* was created, repositioned, and rotated to match the position and orientation the gun should have had when grabbed. Hence, the blue axis was in the direction of the barrel, and the green axis was in the direction of the grip.



Then, it was dragged and dropped to the *Attach Transform* attribute so the gun could be correctly grabbed (< *XR Grab Interactable* component).

Step E completed.

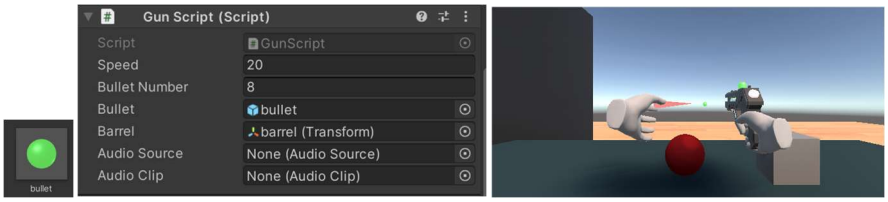
E. The user must be able to pick up the portal gun

8.3. Gun firing

To allow the gun to fire and have a limited number of bullets, a new script was created in the gun object called *GunScript*. A method was developed to allow the gun to fire a limited number of bullets. An audio clip was prepared to be added to make a firing sound in the future.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GunScript : MonoBehaviour
{
    [SerializeField] private float speed = 20; //to set the bullet's speed
    [SerializeField] private int bulletNumber = 8; //number of bullets
    [SerializeField] private GameObject bullet; //the bullet object
    [SerializeField] private Transform barrel; //the barrel of the gun

    [SerializeField] private AudioSource audioSource;
    [SerializeField] private AudioClip audioClip;

    public void Fire()
    {
        if (bulletNumber > 0)
        {
            GameObject spawnedBullet = Instantiate(bullet, barrel.position, barrel.rotation); //new instance of a bullet, to be created in the
barrel position
            spawnedBullet.GetComponent<Rigidbody>().velocity = speed * barrel.forward; //in the direction of the barrel
            bulletNumber = bulletNumber - 1;
            if (audioSource != null && audioClip != null)
            {
                audioSource.PlayOneShot(audioClip);
            }
            Destroy(spawnedBullet, 2); //destroy the bullet after 2 sec
        }
    }
}
```

This method had to be called when the *Activate* event happened on the gun object. A cube, representing the barrel, and a green sphere with a *Rigidbody* component, representing a bullet object, were created in the scene as a child of the gun. Finally, they were assigned to the corresponding script variables.

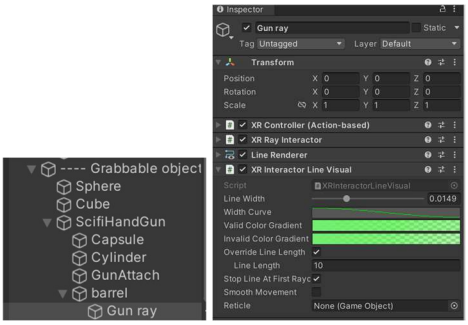


Step F completed.

F. The gun must be able to fire.

8.4. Bullet counter and laser

A *Ray Interactor* was added to the gun (as a child of the barrel) to show a laser in the direction the user was pointing. Thus, the destination of the bullet could be more conveniently known. It was configured to not interact with anything (*Interaction Layer Mask*: *Nothing*).



A bullet counter was added to the gun. To do that, a *Text* was added to the gun (*3D object* > *Text - TextMeshPro*). Moreover, the *TMP Essentials* were imported. The text size, color, and font were changed. It was also positioned. In the *GunScript,* a text variable was added to store the created text. The *Update()* method updated the text value in every frame.

Namespace added:

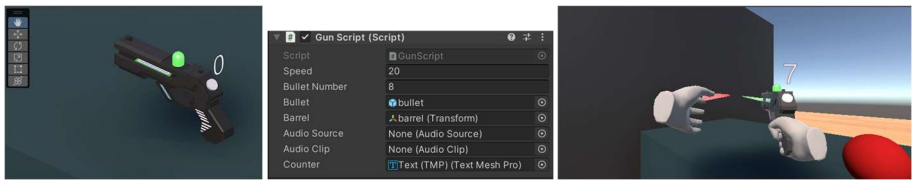
```
using UnityEngine.UI;
using TMPro;
```

Variable added:

```
    [SerializeField] private TMP_Text counter;
```

Method added:

```
    void Update()
    {
        counter.text = bulletNumber.ToString();
    }
```

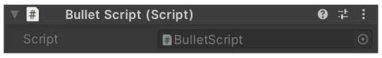


Step G completed.

G. The number of bullets must be limited and there must be a counter.

8.5. Object removal

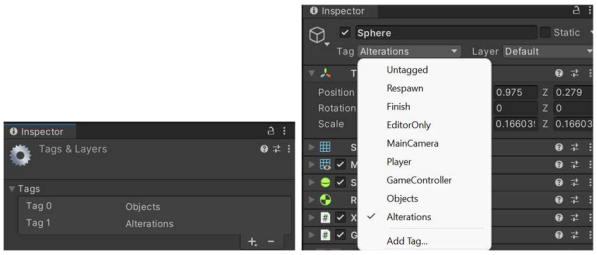
For the bullet to destroy an object when striking it, a script (*BulletScript*) was created and added as a component of the bullet object. The *OnCollisionEnter*() method was used. This method was called whenever the collider/rigid body of the bullet touched another one. The impacted object could be destroyed using the *Destroy()* function with the game object as a parameter.




```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BulletScript : MonoBehaviour
{
    void OnCollisionEnter(Collision collision)
    {
        Destroy(collision.gameObject); //destroy the impacted object
        Destroy(gameObject); //destroy the own bullet
    }
}
```

The bullet could eliminate any impacted entity.

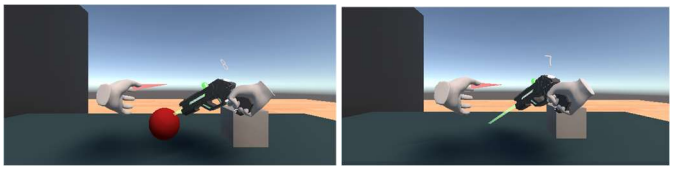
Tags were used to distinguish between elements that could be removed and those that could not. In addition, these tags were used to identify alterations. Therefore, a label called *Alterations* was assigned to the objects that were supposed to be alterations. On the other hand, the *Objects* tag was given to those that could be deleted but were not altered. In the end, the only objects that could not be removed were those corresponding to the room's floor, ceiling, and walls. These elements were *Untagged*. The gun was also untagged, so it could not be removed.



Objects with the *Alteration* or *Object*s tag could be removed when impacted. In addition, the bullet was destroyed when colliding with an object different than the gun. This was because the bullet collided with the gun in the first instance.

Method modified:

```
    void OnCollisionEnter(Collision collision)
    {
        //don't be destroyed when colliding with own gun, it collides with it when fired
        if (impactedObject.gameObject.name != "ScifiHandGun")
        {
            Destroy(gameObject); //destroy the own bullet
        }
        GameObject impactedObject = collision.gameObject;
        if (impactedObject.tag == "Alterations" || impactedObject.tag == "Objects")
        {
            Destroy(impactedObject); //destroy the impacted object
        }
}
```



Hence, every removable object had to be tagged accordingly.

Step H completed.

H. Bullets must be able to make the object they hit disappear.

Step I completed.

I. The program must be able to distinguish between objects that are alterations and those that aren't.

9. Game manager

The game needed a manager to change scenes and exercise control over removed objects. Thus, a new top-level empty object was created in the *Hierarchy* with a script called *GeneralManager*.

9.1. Objects control

First, the script got the number of alters and standard objects. Then, it was prepared for the scene change.

If the number of standard items was reduced by one, the user lost. If the alterations were all removed, it meant that the user won. Afterward, a change of scene to the Game Over scene had to occur.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GeneralManager : MonoBehaviour
{
    private int numAlterationIni; //number of initial alterations
    private GameObject[] alterationsIni; //initial alterations
    private GameObject[] alterations; //alterations update


    private GameObject[] objectsIni; //initial objects
    private GameObject[] objects; //objects update


    // Start is called before the first frame update
    void Start()
    {
        alterationsIni = GameObject.FindGameObjectsWithTag("Alterations"); //get the objects with tag alterations
        objectsIni = GameObject.FindGameObjectsWithTag("Objects"); //get the objects with tag objects
        numAlterationIni = alterationsIni.Length; //store the number of initial alterations, to get the number of bullets
    }

    // Update is called once per frame
    void Update()
    {
        alterations = GameObject.FindGameObjectsWithTag("Alterations"); //get the objects with tag alterations
        objects = GameObject.FindGameObjectsWithTag("Objects"); //get the objects with tag objects

        if (objects.Length != objectsIni.Length) //if some normal object has been removed
        {
            //game over --> player looses
        }
        else
        {
            if (alterations.Length == 0)
            {
                //game over --> player wins
            }
        }
    }

    public int getNumAlterations()
    {
        return numAlterationIni;
    }
}
```

A variable and method were introduced to know the number of initial alterations. This value set the number of bullets according to the number of alters. Thus, the *GunScript* code was modified:

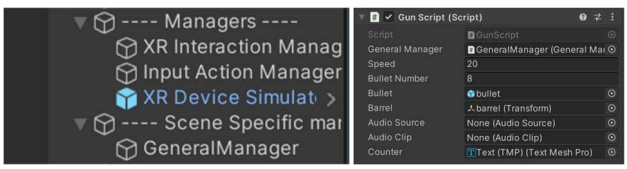
Variable added:

```
    [SerializeField] private GeneralManager generalManager; //to get the general manager
```

Method added:

```
    void Start()
    {
        bulletNumber = generalManager.getNumAlterations();
    }
```

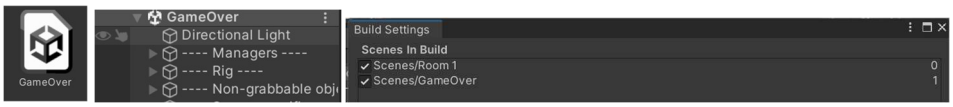
Finally, the object which contained the *GeneralManager* script was dragged to the *GunScript* new variable. The general manager's object was organized in a different category than the other managers as this one was specific to the scene.



9.2. Scene change

The scene change had to be implemented in the *GeneralManager* script.

An empty scene called *GameOver* was created in the *Project* window (*Create > Scene*) without the *Main Camera*. The managers, rig, and plane from *Room 1* scene were copied and added to the scene. Furthermore, the scene was added to the build settings (*File > Build Settings...*).



A ***Static Class*** called *GameResult* was created to hold the game result when changing the scene. This class could not be attached to any *GameObject,* and it would not be destroyed when changing scenes. Moreover, it could be accessed from anywhere.

A static variable to store the result and another one to keep the number of remaining alterations at the end of the game were added. Both of them had their getters and setters.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public static class GameResult
{
    public static int result  { get; set; } // 0 if looses, 1 if wins
    public static int NumAlterations { get; set; } // number of alterations remaining
}
```

Finally, the *Update()* method of the *GeneralManager* script was modified so that the scene change would happen at the end of the game. The *SceneManager.LoadScene()* method was used. When a scene was changed, all current scene objects were destroyed. The *DontDestroyOnLoad(GameObject.Find())* method could have been used to not destroy a specific object.

Namespace added:

```
using UnityEngine.SceneManagement;
```

Method modified:

```
    void Update()
    {
        alterations = GameObject.FindGameObjectsWithTag("Alterations"); //get the objects with tag alterations
        objects = GameObject.FindGameObjectsWithTag("Objects"); //get the objects with tag objects

        if (objects.Length != objectsIni.Length) //if some normal object has been removed
        {
            //game over --> player looses
            GameResult.result = 0;
            GameResult.NumAlterations = alterations.Length;

            //The SceneManager loads the new Scene as a single Scene (not overlapping). This is Single mode.
            Destroy(GameObject.Find("ScifiHandGun"));//because it was giving a non important warning
            SceneManager.LoadScene("GameOver", LoadSceneMode.Single);
        }
        else
        {
            if (alterations.Length == 0 && alterations.Length != numAlterationIni)
            {
                //game over --> player wins
                GameResult.result = 1;
                GameResult.NumAlterations = alterations.Length;

                SceneManager.LoadScene("GameOver", LoadSceneMode.Single);
            }
        }
    }
```

Step J completed.

J. The game must be over if an object that is not an alteration is shot at.

Step K completed.

K. The game must be over if all alters are removed.

Step L completed.

L. Must be able to change scenes.

10. Random disturbances

Although it was not necessary to implement various levels in the game, it was decided to implement a mechanism of randomness in the creation of alters. This way, having only one room designed, the game experience of each game would be different. Moreover, a change of scene to simulate the disturbances was not necessary.

It was implemented in the *GeneralManager* script, and several of the things implemented were changed.

First, some prefabricated were added to a new folder called *SpawnObjects* created inside the *Assets>Resources* folder. These objects had the *Grabbable* tag assigned and were the objects that could be created, grabbed, and altered. The folder was accessed using the *Resources.Load(SpawnObjects)* method. The prefabricated were configured with all the necessary components, tags, and interactable events.

Then, all the grabbable objects from the scene were replaced with 3D cubes tagged as *Grabbable*. All the objects that could be removed using the gun but were not grabbable were tagged as *Objects*.

All the objects needed to be tagged properly.

The script was coded so that at the beginning of the game, all the cubes were replaced by a grabbable object from the folder unexpectedly and with a random color. The cubes indicated a position where a grabbable object could be placed. When an object was replaced, the new object was set with the same position and rotation. Hence, the start of each game was unexpected and different.

Furthermore, a countdown timer was set. A float variable was created to store the amount of time remaining for the alterations. The duration of the previous frame (the delta time) was subtracted from that amount in each frame. When the time ran out, the method assigned the alterations were called. It randomly altered a certain percentage of the grabbable objects changing the color of half of them and destroying and replacing the other half.

Some changes were introduced to *GrabbableScript*. The tag of an object was changed depending on its gripped condition. This was because otherwise when the alterations occurred, if the player held an object and it was altered, it would have been triggered by colliding with the hand.

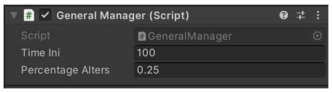
Full Script:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class GeneralManager : MonoBehaviour
{
    private int numAlterationIni; //number of initial alterations
    private int numAlterationActual; //number of actual alterations

    private GameObject[] alterationsIni; //initial alterations
    private GameObject[] alterations; //alterations update


    private GameObject[] objectsIni; //initial objects
    private GameObject[] objects; //objects update

    [SerializeField] private float timeIni = 15;//number of seconds remaining
    [SerializeField] private float percentageAlters = 0.25f;//percentage of alters in

    private bool isAltered = false; //to store if the world has been altered
    private bool isTimerActive = false;

    GameObject[] prefabs;//store all prefabs grabbable

    private int numDestroyed = 0;

    private void Start()
    {
        isTimerActive = true;
        prefabs = Resources.LoadAll<GameObject>("SpawnObjects"); //get all prefabs spawned
        InitializeGame();
    }
    // Update is called once per frame
    void Update()
    {
        if (isTimerActive)
        {
            if (timeIni > 0)
            {
                timeIni -= Time.deltaTime;
            }
            else
            {
                timeIni = 0;
                alterWorld();
                alterationsIni = GameObject.FindGameObjectsWithTag("Alterations"); //get the objects with tag alterations
                objectsIni = GameObject.FindGameObjectsWithTag("Objects"); //get the objects with tag objects
                numAlterationIni = alterationsIni.Length; //store the number of initial alterations, to get the number of bullets

                isTimerActive = false;//so that it doesn't happen in everyframe, only once
            }
        }
        if (isAltered)
```

```
        {
            if (GameFinished())
            {
                //The SceneManager loads the new Scene as a single Scene (not overlapping). This is Single mode.
                Destroy(GameObject.Find("ScifiHandGun"));//because it was giving a non important warning
                SceneManager.LoadScene("GameOver", LoadSceneMode.Single);
            }
        }
        numAlterationActual = GameObject.FindGameObjectsWithTag("Alterations").Length;
        GameResult.NumAlterations = numAlterationActual;
    }

    private void alterWorld()
    {
        GameObject[] grabbableObjectArr = GameObject.FindGameObjectsWithTag("Grabbable"); ;//get the grabbable objects

        //produce a % of alterations randomly assigned
        int alterations = Mathf.RoundToInt(grabbableObjectArr.Length*percentageAlters);//% of objects will be alterations
        int colorAltered = (int) alterations / 2;//half of the alterations will be changed of color
        int destroyAltered = alterations - colorAltered;//the rest will be destroyed and replaces

        List<int> colorAlteredIndexs = new List<int>();//store the index of the color altered objects, initially empty
        List<int> destroyAlteredIndexs = new List<int>();//store the index of the destroy altered objects, initially empty


        for (int i = 0; i < colorAltered; i++) //get the index of color altered object
        {
            bool indexValid = false;
            while (!indexValid)
            {
                int index = Random.Range(0, grabbableObjectArr.Length);// select randomly one grabbable object
                if (!colorAlteredIndexs.Contains(index))
                {
                    colorAlteredIndexs.Add(index);//add the index object as altered
                    changeObjectColor(grabbableObjectArr[index]); //call the function that changes its color and tag
                    grabbableObjectArr[index].tag = "Alterations";//change its tag to alteration
                    indexValid = true;
                }

            }
        }

        for (int i = 0; i < destroyAltered; i++) //get the index of destroy altered object
        {
            bool indexValid = false;
            while (!indexValid)
            {
                int index = Random.Range(0, grabbableObjectArr.Length);// select randomly one grabbable object
                if (!destroyAlteredIndexs.Contains(index) && !colorAlteredIndexs.Contains(index)) //if not already altered
                {
                    destroyAlteredIndexs.Add(index);//add the index object as altered
                    GameObject newObject = destroyObject(grabbableObjectArr[index]);
                    newObject.tag = "Alterations";//change its tag to alteration
                    numDestroyed++;
                    indexValid = true;
                }

            }
        }
        isAltered = true;
        //get all the rest of grabbable objects and change its tag to objects
        GameObject[] grabbableArrAfterAlt = GameObject.FindGameObjectsWithTag("Grabbable"); ;//get the grabbable objects
        foreach (GameObject obj in grabbableArrAfterAlt)
        {
            obj.tag = "Objects";
        }
    }
    private void changeObjectColor(GameObject alter)
    {
        // apply color change on the object's material and childs
        Transform[] childs = alter.GetComponentsInChildren<Transform>(); //list containing the object and its childs
        float r = Random.value;
        float g = Random.value;
        float b = Random.value;

        foreach (Transform child in childs)
        {
            try
            {
                if (child.GetComponent<Renderer>() != null)
                {
                    float rC = child.GetComponent<Renderer>().material.color.r;
                    float gC = child.GetComponent<Renderer>().material.color.g;
                    float bC = child.GetComponent<Renderer>().material.color.b;
                    // pick a random color
                    Color newColor = new Color((rC + r)%1, (gC + g) % 1, (bC + b) % 1);
                    child.GetComponent<Renderer>().material.color = newColor;//change the material color, in unity no problem if exceeds 1.0f
                }
            }
            catch
            {
                continue;
            }

        }
    }

    private GameObject destroyObject(GameObject alter)
    {
        //get one random prefab from the folder
        GameObject prefabRandom = prefabs[Random.Range(0, prefabs.Length)];
        try
        {
            if (alter.GetComponent<GrabbableScript>().isObjectGrabbed()) //if object is grabbed don't destroy
            {
                changeObjectColor(alter);
                numDestroyed--;
                return alter;
            }
        }
        catch
        {
        }
        //create new object in same  position
        GameObject newAlter = Instantiate(prefabRandom, alter.transform.position, alter.transform.rotation);
        //change its color
        changeObjectColor(newAlter);
        Destroy(alter);//destroy the other
        return newAlter;
```

```
    }

    private void InitializeGame()
    {
        GameObject[] grabbableObjectArr = GameObject.FindGameObjectsWithTag("Grabbable"); ;//get the grabbable objects parent

        foreach (GameObject grab in grabbableObjectArr)
        {
            GameObject newObj = destroyObject(grab);
            newObj.tag = "Grabbable"; //just in case they don't have it
        }
    }

    private bool GameFinished()
    {
        alterations = GameObject.FindGameObjectsWithTag("Alterations"); //get the objects with tag alterations
        objects = GameObject.FindGameObjectsWithTag("Objects"); //get the objects with tag objects

        if (objects.Length < (objectsIni.Length - numDestroyed)) //if some normal object has been removed
        {

            //game over --> player looses
            GameResult.result = 0;
            return true;
        }
        else
        {
            if (alterations.Length == 0 && alterations.Length != numAlterationIni)
            {
                //game over --> player wins
                GameResult.result = 1;
                return true;
            }
        }
        return false;
    }

    public int getNumAlterations()
    {
        return numAlterationActual;
    }
    public bool isWorldAltered()
    {
        return isAltered;
    }
```

Each object that could be created had to be in the *SpawnObjects* folder.



On the other hand, the gun did not get the number of bullets according to the number of alters produced. It was configured to get the number of alterations in the game's first frame. Thus, the *GunScript Update()* method was modified. The value changed when the world alteration occurred and only once.

Variable added:

```
    private bool bullIsSet = false;
```

Methods modified and added:

```
    void Update()
    {
        if (generalManager.isWorldAltered() && !bullIsSet)//if the world is altered
        {
            bulletNumber = generalManager.getNumAlterations();//set the initial amount of bullets
            bullIsSet = true; //do it only once
        }
        counter.text = bulletNumber.ToString();
    }
```

Finally, the remaining time for the alterations in the game was shown. A simple *Text* was created in a canvas as described in section 11. The *GeneralManager* script included a method to convert the remaining time into minutes and seconds. This method, called in the *Update() function,* changed the text content accordingly.

Namespace added:

```
using TMPro;
```

Variable added:

```
    [SerializeField] private TMP_Text TimeText;//to display the time in the scene
```

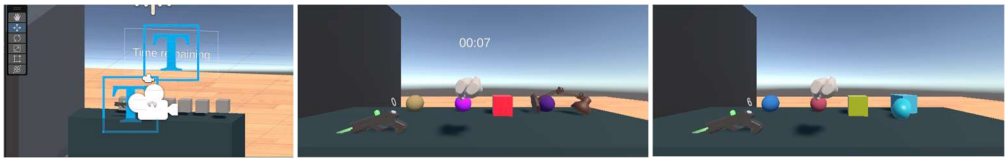
Methods modified and added:

```
    void Update()
    {
        if (isTimerActive)
        {
            if (timeIni > 0)
            {
                timeIni -= Time.deltaTime;
            }
            else
            {
```

```
                timeIni = 0;
                alterWorld();
                alterationsIni = GameObject.FindGameObjectsWithTag("Alterations"); //get the objects with tag alterations
                objectsIni = GameObject.FindGameObjectsWithTag("Objects"); //get the objects with tag objects
                numAlterationIni = alterationsIni.Length; //store the number of initial alterations, to get the number of bullets

                isTimerActive = false;//so that it doesn't happen in everyframe, only once
            }
        }
        if (isAltered)
        {
            if (GameFinished())
            {
                //The SceneManager loads the new Scene as a single Scene (not overlapping). This is Single mode.
                Destroy(GameObject.Find("ScifiHandGun"));//because it was giving a non important warning
                SceneManager.LoadScene("GameOver", LoadSceneMode.Single);
            }
        }
        numAlterationActual = GameObject.FindGameObjectsWithTag("Alterations").Length;
        GameResult.NumAlterations = numAlterationActual;
        DisplayTime(timeIni);
    }

    private void DisplayTime(float time)
    {
        float minutes = Mathf.FloorToInt(time / 60);
        float seconds = Mathf.FloorToInt(time % 60);
        TimeText.text = string.Format("{0:00}:{1:00}", minutes, seconds);
        if (timeIni == 0)
        {
            Destroy(GameObject.Find("Canvas Time Text"), 1);//destroy the canvas in 1 second after timer finishes
        }
    }
```

Finally, the text object was assigned to the script variable.

New objects were added to evaluate their correct functioning, and all of them were altered.



Step M completed.

M. The game must create the alters randomly at the start of each game.

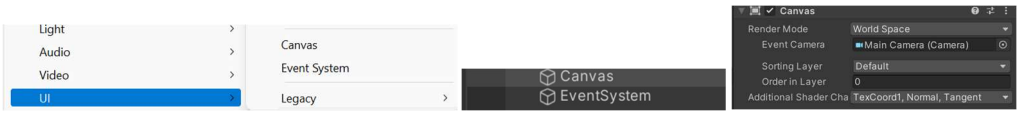
Step N completed.

N. There must be a timer.

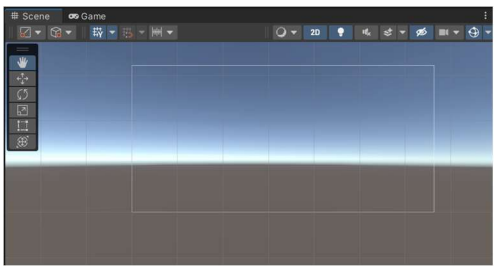
11. User interface (UI)

A UI was created in the Game Over scene to show the result. Moreover, a button was created to allow the user to restart the game. The rays were allowed to interact with the UI.

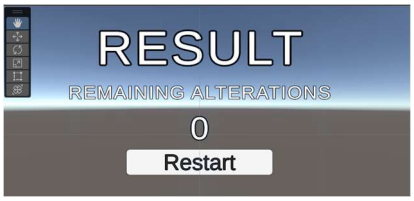
Thus, a simple *UI Canvas* (*UI > Canvas*) was created in the *Hierarchy*. This added a *Canvas* and an *Event System* object, enabling interaction with the UI. The *Canvas Render Mode* property was set to *World Space,* and the *XR Rig Main Camera* was selected as the *Render Camera* so that the *Canvas* was part of the world.



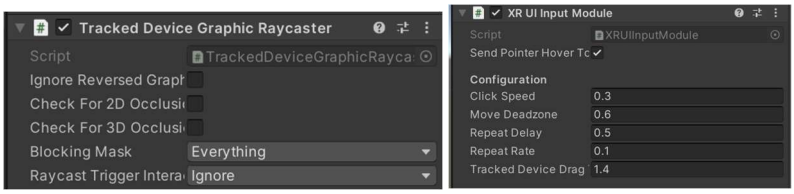
The canvas was positioned in the world. The editor was set to 2D view to work more comfortably.



Some elements were added, positioned, and configured (*UI > select element*). Specifically, a text to display the result, a text to indicate the number of alterations remaining, and a button to restart the game.



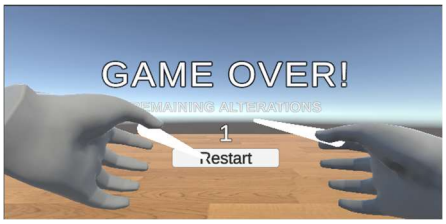
A *Tracked Device Graphic Raycaster* component was added to the canvas. In the *EventSytem*, the *Standalone Input Module* component was removed. This component used the old *InputManager*. Then, the *XR UI Input Module* component was added to allow correct interaction with the UI.



Finally, a new empty object with a script called *UIManager* was created. This script changed the text of the user interface depending on the result and had a method to switch to the game scene when clicking on the restart button.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using TMPro;

public class UIManager : MonoBehaviour
{
    [SerializeField] private TMP_Text result;
    [SerializeField] private TMP_Text altersRemainingNum;
    private string[] resultText = {"GAME OVER!","CONGRATULATIONS!"};

    // Start is called before the first frame update
    void Start()
    {
        result.text = resultText[GameResult.result];
        altersRemainingNum.text = GameResult.NumAlterations.ToString();
    }

    public void restartGame()
    {
        SceneManager.LoadScene("Room 1", LoadSceneMode.Single);
    }
}
```

The corresponding objects were assigned to the variables. In the button *OnClick()* event, the *restartGame()* method was called.



All the functions of the game were now operational. All that remained was to design and make the scene changes when necessary.

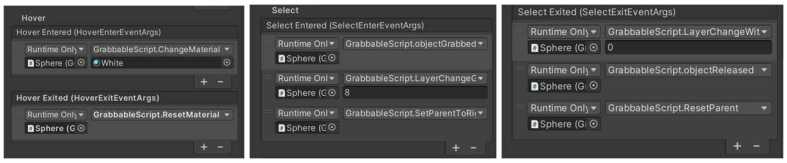
12. Scene design

All the necessary elements for the application operation were set up; the only thing left was to design the scenes.

12.1. Game room design

In designing the game room scene, specific considerations were considered:

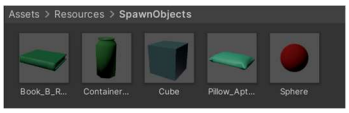
- All objects that could not be grabbed and that should not be interacted with had to be *Untagged*.
- All objects that could not be grabbed and that should be interacted with (can be destroyed) had to be tagged with the *Objects* tag.
- All objects representing a grabbable object in the scene (cubes) had to be tagged with the *Grabbable* tag.
- All objects that could be grabbed:
    - Needed a *RigidBody* component with *Collision Detection* set to *Continuous Dynamic*.
    - Needed the *GrabbableScript*.
    - Needed the *XR Grab Interactable* component with:
        - *Movement Type* to *Velocity tracking*.
        - Interactable events set: *Hover Entered*, *Hover Exited*, *Select Entered*, and *Select Exited*.
- All objects that could be grabbed but not spawned:
    - Needed to be tagged with the *Objects* tag.
- All objects that could be spawned:
    - Needed to be grabbable.
    - Needed to be tagged with the *Grabbable* tag.
    - Needed to be added to the *Resources > SpawnObjects* folder.



The game room was created using prefabricated and assets from the Asset Store and the Unity Learn project. Some of the prefabricated used the *Standard Shader* and were pink colored. It was solved by going to *Window > Rendering > Render Pipeline Converter* and converting all the assets from *built-in* to *URP*.

Some objects were introduced into the *SpawnObjects* folder. It was a limited amount of time to make the game less challenging. In the scene, cubes were placed. Objects from the SpawnObjects folder replaced these cubes at the beginning of the game.

The objects in the *SpawnObjects* were cans, pillows, cubes, spheres, and books:



The number of bullets in the weapon was set to unlimited because it was difficult to hit an object from a distance. The counter still showed the number of alters left to be eliminated. In addition, the gun was only allowed to fire if alterations existed.

Full GunScript:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using TMPro;

public class GunScript : MonoBehaviour
{
    [SerializeField] private GeneralManager generalManager; //to get the general manager

    [SerializeField] private float speed = 20; //to set the bullet's speed
    [SerializeField] private int bulletNumber; //number of bullets
    [SerializeField] private GameObject bullet; //the bullet object
    [SerializeField] private Transform barrel; //the barrel of the gun

    [SerializeField] private AudioSource audioSource;
    [SerializeField] private AudioClip audioClip;

    [SerializeField] private TMP_Text counter;


    void Update()
    {
        bulletNumber = generalManager.getNumAlterations();//set the amount of bullets
        counter.text = bulletNumber.ToString();
    }

    public void Fire()
    {
```

```
        if (bulletNumber > 0) //only able to fire if there are alterations
        {
            GameObject spawnedBullet = Instantiate(bullet, barrel.position, barrel.rotation); //new instance of a bullet, to be created in the
barrel position
            spawnedBullet.GetComponent<Rigidbody>().velocity = speed * barrel.forward; //in the direction of the barrel
            if (audioClip != null && audioSource != null)
            {
                audioSource.PlayOneShot(audioClip);
            }
            Destroy(spawnedBullet, 2); //destroy the bullet after 2 sec
        }
    }
}
```

Finally, a start button was added to the scene in case the player was ready to find the alterations before the timer ended. Thus, a new function in the *GeneralManager* was added. This function set the timer to zero and was called when clicking the button.

Method added:

```
    public void endTimer()
    {
        timeIni = 0;
    }
```

Moreover, a canvas to display the current system time was added.

Step O completed.

O. The game room scene must be fully designed.

12.2. Game Over scene design

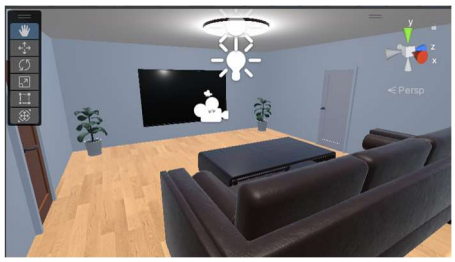
To design the game over scene, the UI was changed. In addition, the game room was duplicated and filled with some objects. It is important to notice that the player could not interact with the objects, only with the UI.

Step P completed.

P. The game over scene with User Interface (UI) must be created.

12.3. Start Menu scene design

A start menu scene was created with UI. It was designed as a room where start and tutorial buttons appeared in the TV area. The start button took the player to the game scene and the tutorial button to the tutorial scene.



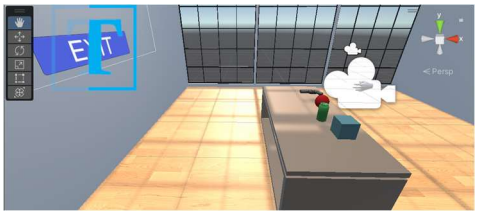
Step Q completed.

Q. The start menu scene must be created.

12.4. Tutorial scene design

This scene was intended to be used as a tutorial. Therefore, it was designed as a game room with just three objects of assorted colors and a gun. The steps the user had to follow to understand how the game worked were shown in a UI that followed the camera. In addition, the user had the option to quit the tutorial.

The first step for the player was to pick up an object. Then the player had a certain amount of time to remember the three objects and their colors. When the time finished, some disturbances occurred, and the user had to eliminate them using the gun to complete the tutorial.



Step R completed.

R. Create a tutorial.

13. Extras

13.1. Particles

Some particles were created when an object was destroyed. Thus, a *Particle System* was added as a game object to the project (*GameObject > Effects > Particles System*). The particles were configured with a sphere shape. The material, start lifetime, and speed were set.

Once created and configured, the particles were removed from the *Hierarchy* window and dragged to the *Prefabs* folder inside the *Project* window. Finally, they were created at the object's world position when destroyed. This code was added to the *BulletScript* and the particles to the script component variable inside the bullet object.

13.2. Sounds

In addition, some sounds were added to the game.

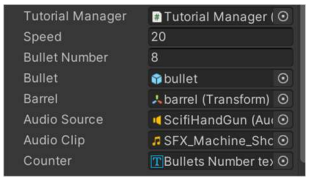
An empty *GameObject* containing an *AudioSource* component was created to play a sound when an alteration was removed. Another one was created to play a different sound when an entity that is not an alteration was removed.

The corresponding clip was added to the audio source component, and the *Play on Awake* property was enabled. The corresponding object was instantiated in the *BulletScript* when destroying an object and removed after one second. It would have been more convenient to put the component directly into the bullet object containing the script. However, this was impossible as the bullet was destroyed, and destroying a *GameObject* would have destroyed any part attached to it. The *AudioSource* component would have been destroyed when the bullet was destroyed, and the sound would not have been played.

Full BulletScript:

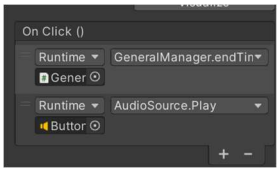
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BulletScript : MonoBehaviour
{
    [SerializeField] private GameObject particles; //the particles object
    [SerializeField] private GameObject destroyPositive;
    [SerializeField] private GameObject destroyNegative;


    void OnCollisionEnter(Collision collision)
    {
        GameObject impactedObject = collision.gameObject;
        if (impactedObject.tag == "Alterations" || impactedObject.tag == "Objects")
        {
            Destroy(impactedObject); //destroy the impacted object
            //create the particles in the objects position
            Instantiate(particles, impactedObject.transform.position, impactedObject.transform.rotation);
            if (impactedObject.tag == "Alterations")
            {
                GameObject destroySound = Instantiate(destroyPositive);
                Destroy(destroySound, 1);


            }
            else
            {
                GameObject destroySound = Instantiate(destroyNegative);
                Destroy(destroySound, 1);
            }
        }
        //don't be destroyed when colliding with own gun, it collides with it when fired
        if (impactedObject.gameObject.name != "ScifiHandGun")
        {
            Destroy(gameObject); //destroy the own bullet
        }
    }
}
```

On the other hand, an *AudioSource* component was added to the gun and dragged to the corresponding variable of the *GunScript* component. The trigger sound was added to the audio clip variable to produce the trigger sound when the gun fired.



Furthermore, sounds were added to the UI buttons in the different scenes. Audio source objects were created in the *Project* window with the *Play on Awake* property disabled and the corresponding audio clips.

For the button in the game room, an event was created. It called the *AudioSource.Play* function. The game object was added to the scene. The initial idea was to do the same for every button. However, since the other buttons made a load of a set, it gave problems. The corresponding object was instantiated to solve it, and the play function was called in the on-click event method.



The sounds used until this point were from the Unity Learn FX sounds folder.

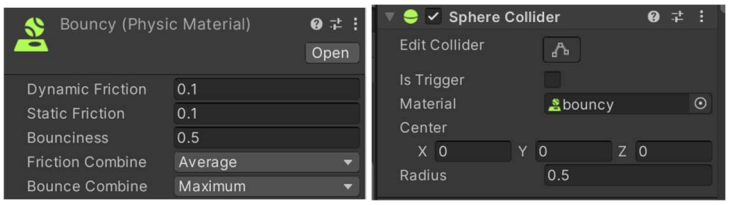
Then, more sounds were added to the scenes following similar processes. Audio clips with a *Creative Commons 0* license from the *freesound.org* webpage, an initiative of *UPF*, were added.

In the game-over scene, a different sound was added depending on the game's result. A sound was added when disturbances occurred in the game room and tutorial scenes. Finally, a sound was configured to be played when an object was grabbed.

13.3. Object configuration

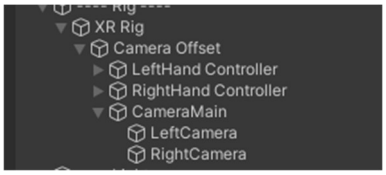
Finally, the size of the objects was adjusted according to the real-world scale for a better VR experience. Also, their mass was set according to the behavior desired when interacting with the player. Some objects were made heavier than others. This was established in the *RigidBody* component of each object.

On the other hand, *Material Physics* was added to the spheres to provide bouncy properties. A new physics material with bouncy properties was created (*create > Physics Material*) and assigned to the material property of the spheres object's *Sphere Collider* component.

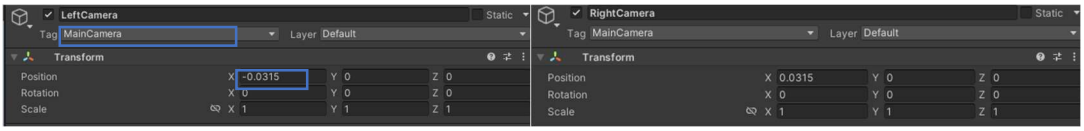


13.4. Stereoscopic-View

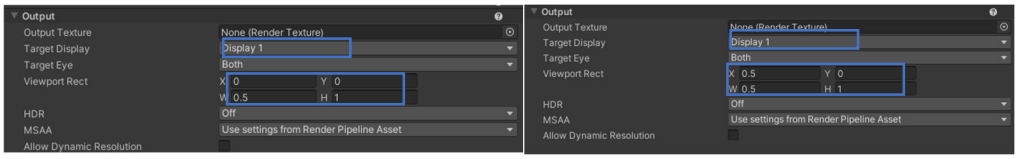
To implement a stereoscopic or split view in the VR application (necessary for the experiments), two camera objects were added as child objects to the main camera, one for each eye.



The camera objects were tagged as "MainCamera" to ensure proper functionality as game cameras in Unity. To create the stereoscopic effect, the left and right eye cameras were positioned 31.5mm to the left and right, respectively, resulting in an inter-pupillary distance (IPD) of 63mm.

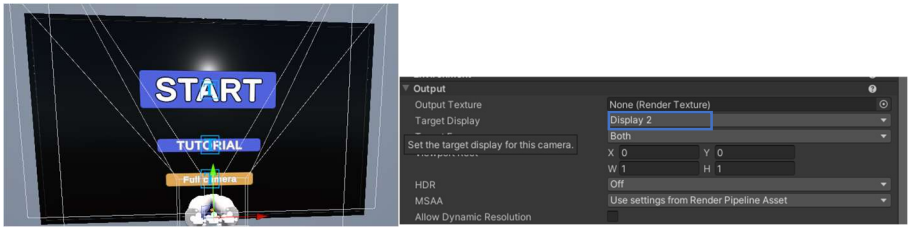


Subsequently, the width of both cameras was halved, and the right camera's viewport position was adjusted by moving it 0.5 units to the right. The target display for both cameras was set to "Display 1" to ensure they were correctly displayed in the game view.



Left Right

To provide flexibility, a button was added to the main menu UI canvas to enable switching between a full-screen view and a split view. This was accomplished by modifying the target display of the "CameraMain" (full camera view) object to "Display 2" initially, ensuring it remained hidden.



When the button was clicked, the UIManagerMenu script's toggleGameView() method was invoked. This method updated the button text and adjusted the cameras' target displays accordingly. Hence to hide a camera it was set to Display 2 (index 1) and to show it to Display 1 (index 0).Additionally, an integer value, acting as a Boolean, was added to the GameResult script to indicate the selected view. Hence, evey

Added method to UIManagerMenu:

```
//method to change game display view
    public void toggleGameView()
    {
        ButtonClick.GetComponent<AudioSource>().Play();
        if (GameObject.Find("Display button") != null)
        {
            GameObject displayBut = GameObject.Find("Display button");
```

```
            GameObject displaytxt = displayBut.transform.GetChild(0).gameObject;
            TMP_Text displayEditableTxt = displaytxt.GetComponent<TMP_Text>();
            if (GameObject.Find("CameraMain") != null && GameObject.Find("LeftCamera") != null && GameObject.Find("RightCamera") != null)
            {
                if (displayEditableTxt.text == "Full camera")
                {
                    displayEditableTxt.text = "Left/Right camera";
                    GameObject.Find("CameraMain").GetComponent<Camera>().targetDisplay = 0;
                    GameObject.Find("LeftCamera").GetComponent<Camera>().targetDisplay = 1;
                    GameObject.Find("RightCamera").GetComponent<Camera>().targetDisplay = 1;
                    GameResult.camera = 0;
                }
                else
                {
                    displayEditableTxt.text = "Full camera";
                    GameObject.Find("CameraMain").GetComponent<Camera>().targetDisplay = 1;
                    GameObject.Find("LeftCamera").GetComponent<Camera>().targetDisplay = 0;
                    GameObject.Find("RightCamera").GetComponent<Camera>().targetDisplay = 0;
                    GameResult.camera = 1;
                }
            }

        }

    }
```

To maintain consistent camera settings when transitioning between scenes, the start method in each scene included code to properly set the target display of the cameras based on the value stored in the GameResult camera variable.

14. VR System Deployment

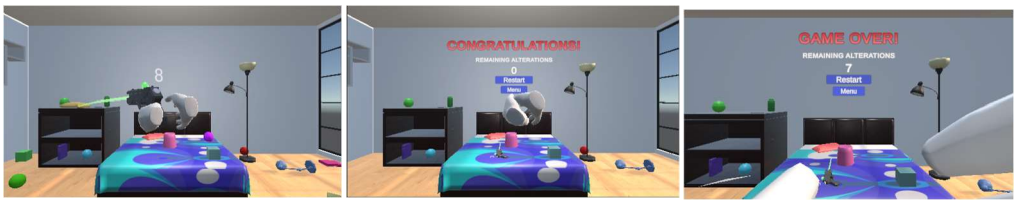
Once the game was developed and validated in the Unity simulator, it was necessary to evaluate its correct performance on VR devices. Tests were conducted using Oculus Air Link, Oculus Link, and by building an APK for VR headsets and Android devices.

To ensure compatibility with these devices, the CameraMain object required the addition of the Tracked Pose Driver component. This step was crucial, as using the Tracked Pose Driver (Input System) alone resulted in improper headset positioning. For Oculus Link and Air Link, connecting the devices to the PC via the Oculus application was sufficient, given that OpenXR was already employed, and all devices were targeted accordingly.

Note that for the APK build, the split view screen option was removed. The steps were following the Oculus guidelines, as the tested headset was an Oculus Quest 2 device, available on: https://developer.oculus.com/documentation/unity/unity-build/. Instead of selecting Oculus Quest 2 as the run device, the option of all available headsets was selected.

Results

This section shows screenshots of the developed game and how it worked. The game worked correctly in the simulator. Disturbances occurred, and the player could use the gun to remove them with all operational functions.



Game future steps

The game may have quite a few improvements to be developed in the future. These include the development of more rooms and the implementation of a level model. In addition, a complete player character could be added, as now, the player is represented simply by hands.

Sockets could also be implemented, and movable furniture such as drawers or cabinets could be added.

In addition, more sounds and particles could be added, and even a game story trailer could be created to present the game.

Unity Render Streaming

Unity Render Streaming is a solution that allows to develop a peer-to-peer streaming solution taking advantage of WebRTC. It allows to broadcast video rendered on Unity to browsers via network. Moreover, the mouse, keyboard, touch, and gamepad can be used as input devices on the browser.

It supports the following browsers:

Browser	Windows	Mac	iOS	Android
Google Chrome	✅	✅		✅
Safari		✅	✅	
Firefox	✅			
Microsoft edge (Chromium based)	✅			

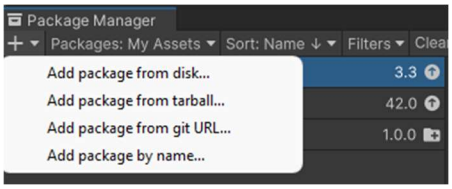
(source: https://docs.unity3d.com/Packages/com.unity.renderstreaming@3.1/manual/index.html)

To develop a streaming application, the following steps were followed:

1. Package installation

The first step was installing the package to start working.

Thus, in the *Window* menu bar, *Package Manager* option was selected and in the *Package Manager* window, the + button was clicked and the *Add package from git URL….* option was selected.



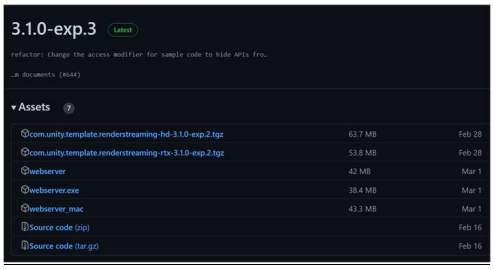
Then, the following string was added to the input field: *com.unity.renderstreaming@3.1.0-exp.4* and after clicking the *Add* button, the package was installed.

2. Launching the web application

First, Node.js was downloaded from https://nodejs.org/en/download/ and installed in order to run the web server's source code in case it was desired to customize the web app.

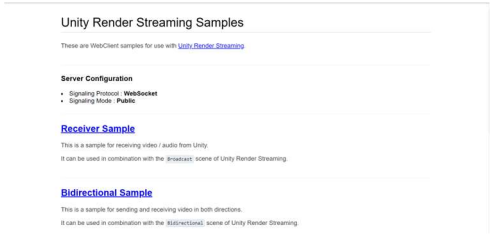
Then, the signaling server was downloaded. To download the web application, I downloaded the webserver.exe file from https://github.com/Unity-Technologies/UnityRenderStreaming/releases. I downloaded the version 3.1.0-exp.3 because exp.4 version of the server had a problem with the verification of the web page path.

Following the Render Streaming 3.1.0-exp.4 manual by clicking on *Edit > Render Streaming > Download web app* menu item on Unity Editor did not work.



After the download finished, a PowerShell window was opened in the folder, and the following command was written to launch it using public mode and web socket signaling: *.\webserver.exe -w*

The server was up and running in the URLs shown on the command line:

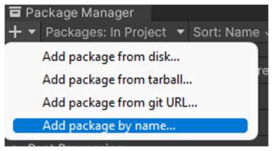


The signaling between Unity and the web browser is managed by the web application, which also serves as the web page's location.

3. Installing WebRTC

Additionally, WebRTC was installed.

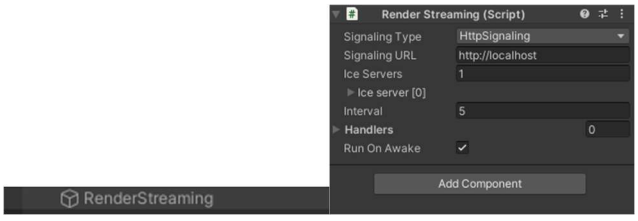
In the *Window* menu bar, *Package Manager* was selected and in the *Package Manager* window, the *+* button was clicked and the *Add package by name…* option was selected.



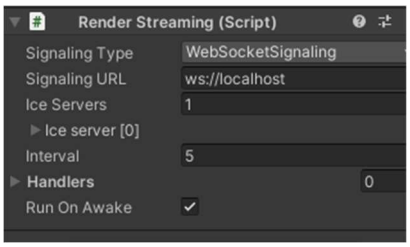
Then, the following string was added to the input field: *com.unity.webrtc* and after clicking the *Add* button, the package was installed.

4. Video streaming

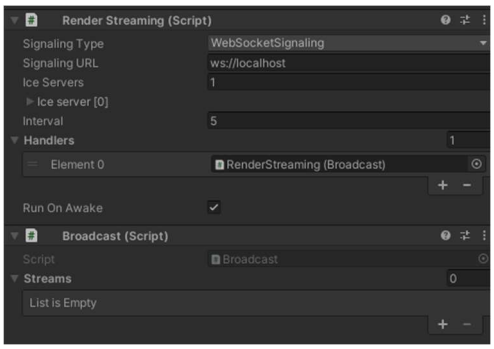
To display the image rendered from the camera to the browser, an Empty object called RenderStreaming was created. It was created in a folder inside the *Resources* folder to be created only at the start of the game. To this object, the *Render Streaming* component was added in the Inspector window.



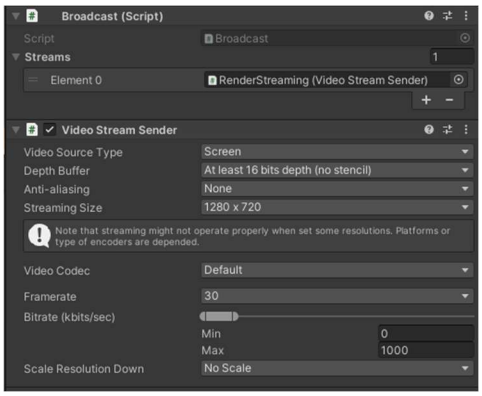
The *Signaling Type* property was set to *WebSocketSignaling* and the *Signaling URL* to *ws://localhost.*



Then, the *Broadcast* component was added to object to deliver the stream to multiple peers. Furthermore, the *Broadcast* component was assigned to the *Handler* property of the *Render Streaming* component.



After, a *Video Stream Sender* component was added to the object. This component refers to the camera and delivers it as a stream to other peers. Some streaming properties such as the streaming size, anti-aliasing, bit rate and framerate can be set which could affect performance. The *Video Stream Sender* component was assigned to the *Broadcast* component property.



Then, the object was made to be instantiated at the beginning of the start menu scene. It is created only the first time the game is started, so that each time the player returns to the menu scene, the *RenderStreaming* object is not created again which could cause problems.

Method changed in UIManagerMenu.cs:

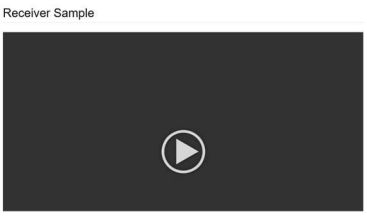
```
    void Start()
    {
        if (GameResult.started == 0)
        {
            renderStreaming = Resources.LoadAll<GameObject>("RenderStreaming");
            Instantiate(renderStreaming[0], transform.position, transform.rotation);
            GameResult.started = 1; //game has been started so no need to create again the RenderStreaming object, if not done when going back to
the menu, two renderstreaming objects would be created
        }

    }
```

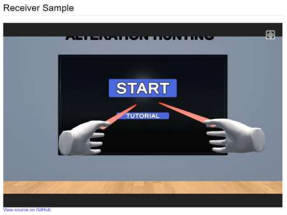
Moreover, the object was kept at every scene change to avoid the connection being closed when changing and being necessary to hit the play button in the browser again. Thus, in the scenes manager scripts where the change of scene was produced, the *DontDestroyOnLoad(GameObject.Find())* method was used to not destroy the *RenderStreaming* object.

Lines added to startGame() and tutorial() methods in UIManagerMenu.cs, to exit() and step5() methods in TutorialManager.cs, to update() method in GeneralManager and to restartGame() and goToMenu() methods in UIManager:

```
if (GameObject.Find("RenderStreaming(Clone)") != null)
        {
            DontDestroyOnLoad(GameObject.Find("RenderStreaming(Clone)"));
        }
```

Finally, after running the game on the editor, going to the *Receiver Sample* page of the web server, and clicking to the play button, the video from the Unity game screen was streamed to the supported browsers and platforms.





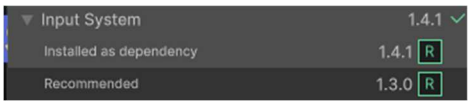
However, the game could be controlled from the Unity Editor but not from the browser.

5. Device input

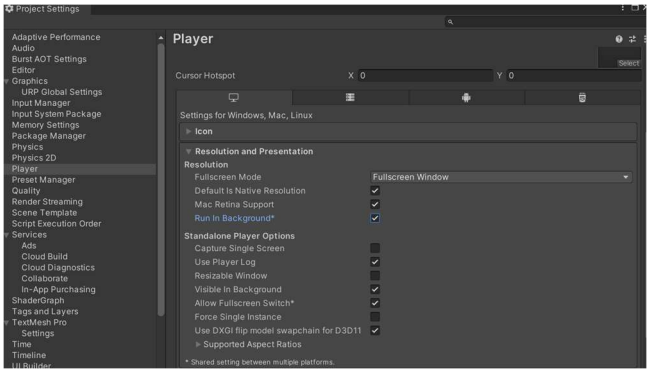
To control the game on a remote PC, the input events on web browsers were linked to the actions in Unity.

Since Unity Render Streaming supports controlling device input using Input System, **Input System** was used to implement the input processing.

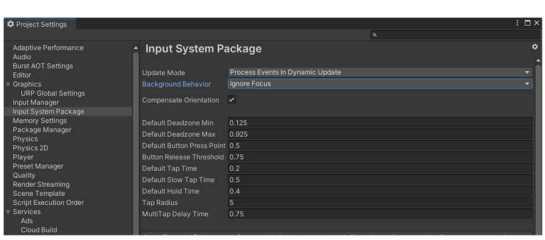
Input system was already installed. In case it had not been installed it could be done by opening the Unity's package manager (*Window > Package Manager*), searching for it in the *Unity Registry packages* list and clicking on *Install*.

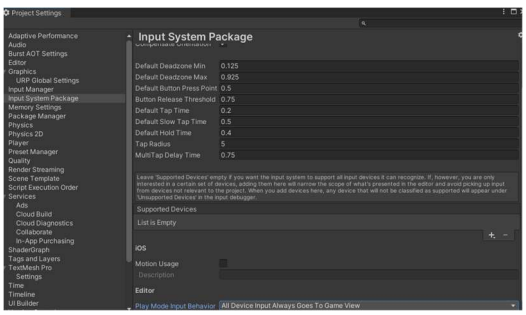


Then, some settings were configured to use Input System. In *Player > Resolution and Presentation* inside *Project Settings* window, *Run In Background* was enabled.

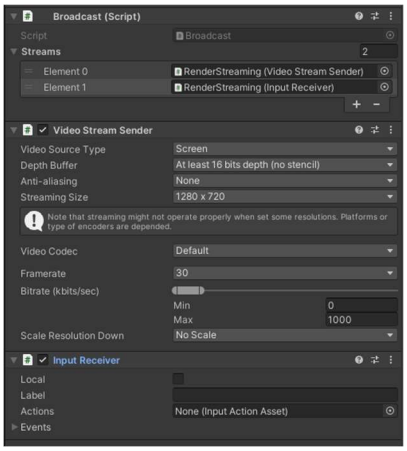


Additionally, in the *Input System Package* tab, *Ignore Focus* for *Background Behavior was set* and *Input Behavior to All Device Input Always Goes To Game View* was set as the *Play Mode*.



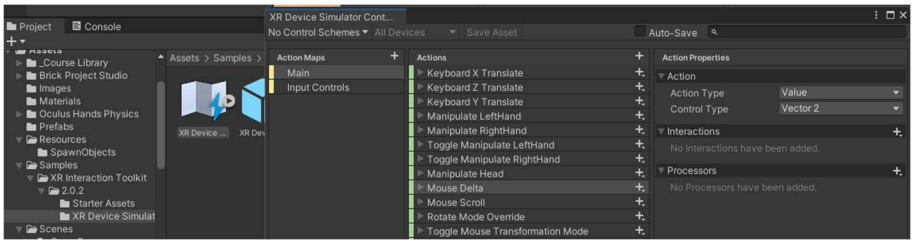


Then, the *Input Receiver* component was added to the *RenderStreaming* objects of all the scenes. And the component was dragged to the *Broadcast* component streams.



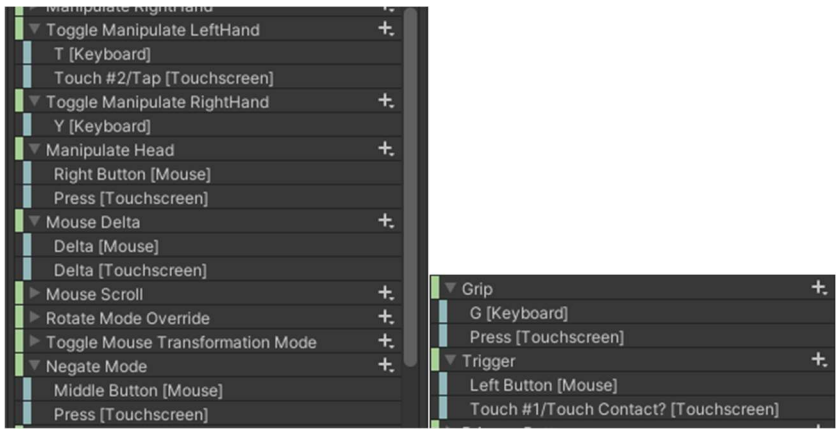
Then, the *Input Actions* feature of the Input System was used. This feature maps various inputs to Unity actions.

As the input system used in the editor simulator was intuitive, the *Input Actions* asset of the *XR Device Simulator* was used.



The asset was modified to allow mobile devices play by using touchscreen inputs. As the game is complex and the number of inputs on a screen is limited, the number of actions was limited so that the game could be played on a touchscreen device. Bindings were added to the Mouse Delta, Manipulate Head, Negate Mode, Toggle Manipulate LeftHand, Trigger and Grip actions.

Thus, the player's head follows the position of the user's fingers, the left-hand manipulation is activated/deactivated by pressing with three fingers, the grip action is achieved by pressing and holding with one finger and the trigger action is produced by tapping with two fingers.

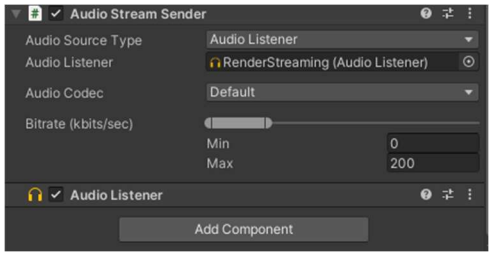


Finally, the *XR Device Simulator Controls* asset was added to the *Actions* property of the *InputReceiver* component of the *RenderStreaming* object.

6. Audio streaming

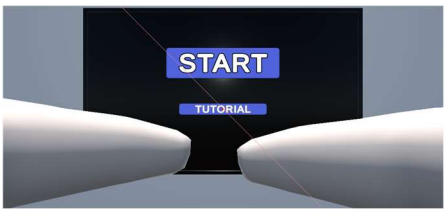
At this point, the game video could be streamed. However, the audio was not streamed.

To do that, an *Audio Stream Sender* component was added to the *RenderStreaming* object and was assigned to the *Streams* property of the *Broadcast* component. Moreover, the *Audio Listener* component in the *Main Camera* object was removed from all the scenes and added to the *RenderStreaming* object as it had to be set to the same object as the Audio Stream Sender. Then, it was assigned to the *Audio Listener* in the *Audio Stream Sender* component.



Thus, the game video and audio could be streamed and controlled from compatible browsers and devices using the keyboard, mouse, and screen.

However, on a device that uses screen input, the gameplay is not great because the user is not able to change the position of the head or hands. This means that when viewing the game, the hands and head are placed together in a start position (0,0,0) as there are no controllers to be tracked. To use it properly on a device with screen input, the camera must first be moved backwards using the mouse. This is not a problem since the game is not intended for such devices specifically.



This could be solved by creating an empty object as a parent of the camera object or both controller objects and changing its position, but that would mean that the VR headset and controllers' positions will not be according to reality. Hence, the VR experience would not be optimal using HMDs.

Thus, the user can play a game which can be streamed to multiple devices and played on some of them.

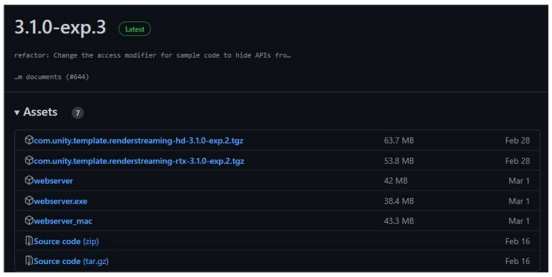
On the other hand, the game could not be controlled using VR Controllers as it was not supported by Unity Render Streaming.

7. Input Data

To capture input data from remote devices for the TFG experiments, an empty object with an InputExport script component was added to the start scene (Main Menu). This object remained intact during scene transitions. The script utilized the InputSystem data and system timestamps to record data from the remote controllers, including the performed actions and associated timestamps. Each streaming session generates a new input event file in a designated folder within the game.

Customizing the web application

The web application was customized. In order to do that the source code of the web server 3.1.0-exp.3 was downloaded from https://github.com/Unity-Technologies/UnityRenderStreaming/releases.



Then, the file was decompressed, and a terminal was open in the *WebApp* folder. The *npm install* command was written to install the dependencies.

```
PS C:\Users\migue\Documents\WebServer\UnityRenderStreaming-3.1.0-exp.3\WebApp> npm install
npm WARN old lockfile
npm WARN old lockfile The package-lock.json file was created with an old version of npm,
npm WARN old lockfile so supplemental metadata must be fetched from the registry.
npm WARN old lockfile
npm WARN old lockfile This is a one-time fix-up, please be patient...
npm WARN old lockfile
npm WARN deprecated w3c-hr-time@1.0.2: Use your platform's native performance.now() and performance.timeOrigin.
npm WARN deprecated uuid@3.4.0: Please upgrade  to version 7 or higher.  Older versions may use Math.random() in certain
 circumstances, which is known to be problematic.  See https://v8.dev/blog/math-random for details.
npm WARN deprecated har-validator@5.1.5: this library is no longer supported
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/request/issues/3142
```

Then, the *npm run build* command was written to build the server and after the *npm run dev -- -w* command was entered to start the WebSocket signaling server.

```
PS C:\Users\migue\Documents\WebServer\UnityRenderStreaming-3.1.0-exp.3\WebApp> npm run build

> webserver@3.1.0 build
> tsc -p tsconfig.build.json
```

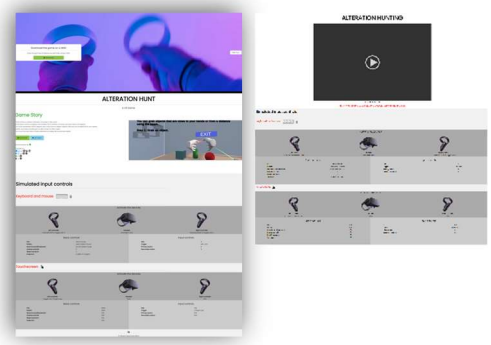
```
PS C:\Users\migue\Documents\WebServer\UnityRenderStreaming-3.1.0-exp.3\WebApp> npm run dev -- -w

> webserver@3.1.0 dev
> ts-node ./src/index.ts -w

start websocket signaling server ws://192.168.3.8
start as public mode
http://192.168.3.8:80
http://127.0.0.1:80
```

The web server was evaluated and worked properly.

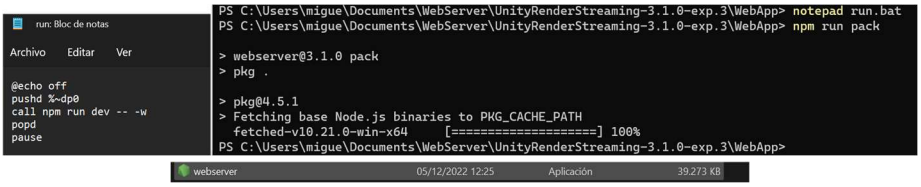
The next step was to modify the web archives to customize the web app to make it simple. These files could be found in the following route *WebApp>client>public.*

The *index.html* file in the public folder which corresponded to the initial html page was modified using a W3 school's template. Moreover the *index.html* file found at *WebApp>client>public>receiver* which contained the receiver sample page was modified too.

The results were the following:



Finally, to facilitate a quick startup, a shortcut was created to run the web server. For this purpose, the run.bat file was modified using the *notepad run.bat* command as it can be seen in the following screenshot. Then, the app was converted into executable binary to cut the effort required to set up the environment with the *npm run pack* command.



To execute the WebSocket signaling server, the following command needs to be used in Windows 11: ***.\webserver.exe -w***.

Comments

It has been a very entertaining project with a satisfactory outcome. It has been complicated due to the little knowledge in the area. One of the most challenging parts has been to keep the game simple because innovative ideas kept coming to my mind, and it wasn't easy to control the impulse to extend the game. It has also been complicated because problems appeared whenever I added new things.

I want to thank **Unity Learn**, a platform with live and on-demand content that has helped me to acquire the necessary knowledge to conduct the project. (https://learn.unity.com/)

I would also like to acknowledge the YouTube channels:

- Valem: https://www.youtube.com/c/ValemVR
- Valem Tutorials: https://www.youtube.com/c/ValemTutorials
- Fist Full of Shrimp: https://www.youtube.com/channel/UCGdxet67QoJij3koOVygssA
- Unity Adventure: https://www.youtube.com/c/UnityAdventure

, which guided me through some steps and helped me solve specific issues.