Interaction Lab

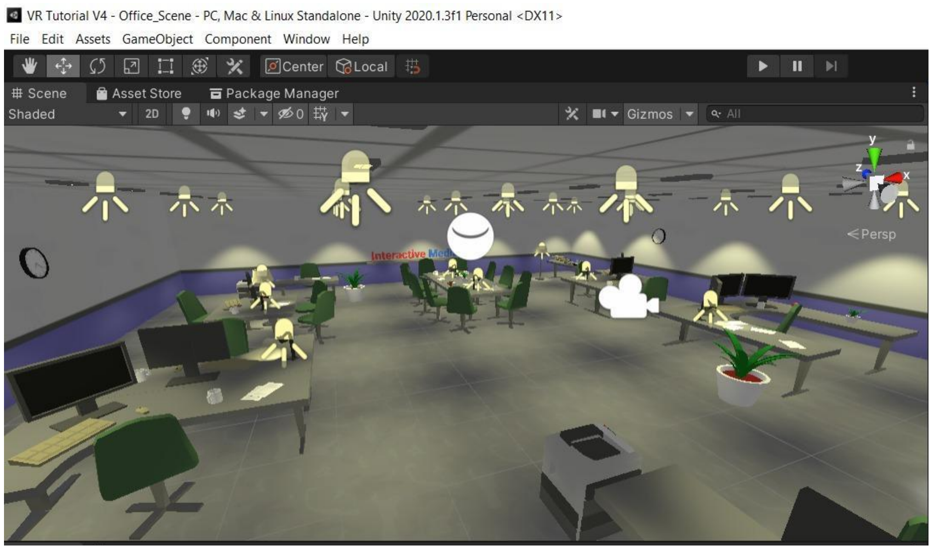
VR Interaction Tutorial

Almost all VR headsets today are compatible with Unity. This tutorial will guide you through setting up your own VR interactions using SteamVR. At the end of the tutorial, you should be able to have a VR player up and running, which can teleport around and grab and throw objects in the scene. The supported headsets are the *Valve Index*, *HTC Vive, HTC Vive Pro, Oculus Rift S, Oculus Quest 1* and *2* (Through Oculus Link), all of which are 6 degrees-of-freedom enabled, meaning they have both rotational and positional tracking. For the tutorial, you will need a working installation of **Unity 2021.1.19f1** or newer.

Headset Configuration

We assume that the Headset in question is already set up, including driver installation. For the *Oculus Rift S*, *Quest 1 and Quest 2* this includes SteamVR and Oculus Drivers, and for the *Quest 1 and Quest 2* specifically, Oculus Link or Oculus Airlink needs to be set up. The *HTC Vive* and *Valve Index* only require SteamVR. Also, the SteamVR Room setup calibration must be run for the *HTC Vive* and *Valve Index* before you can use the device, every time the physical setup is changed.

First, start a new Unity project and download the 'TeleportReticle' folder found on the *Interaction Lab > Unity-xr-basics* GitHub page, and place them in the unity asset folder. (Recommended) Also download the accompanying office environment on the *Interaction Lab > Unity-Assets > Office* GitHub page, and place it in the asset folder. Open the included office environment, located in *Assets > Office > Office_Scene*. Your scene view should now look something like this;

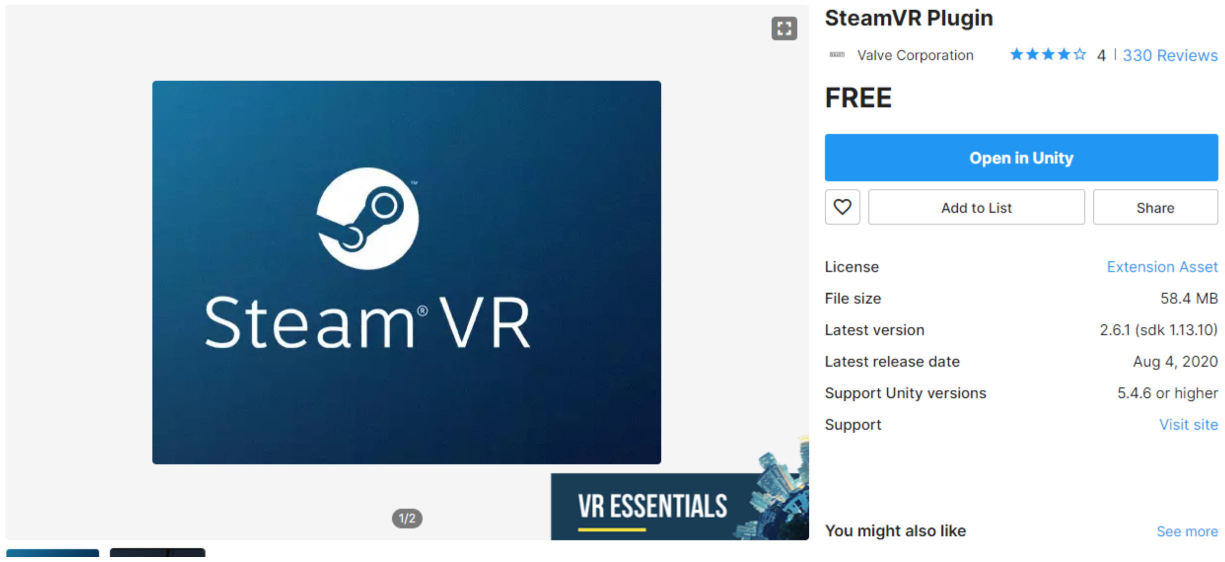

VR Tutorial V4 - Office_Scene - PC, Mac & Linux Standalone - Unity 2020.1.3f1 Personal <DX11>
File Edit Assets GameObject Component Window Help
Center
Local
Scene
Asset Store
Package Manager
Shaded
2D
Gizmos
All
Persp

1. Installing the SteamVR plugin

The first step is to go to the package manager (window > package manager) and select 'XR plugin management (Make sure 'Unity registry' is selected as the packages list on the top). Download and install the newest version.

Then, go to the Unity Asset Store *(click window > asset store, and go through to their website)*, and search for "SteamVR". Find the plugin seen below, press 'add to my assets' and then 'Open in Unity'.

Note: The asset store is a useful tool for other projects as well, here you can find high-quality models and assets which you can add to your project.


SteamVR Plugin

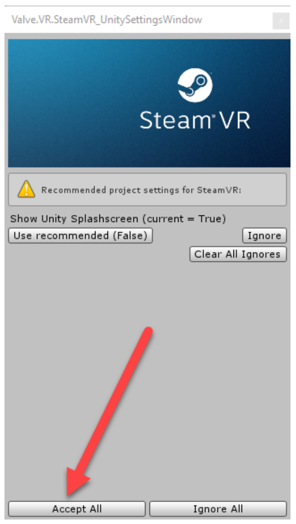
Valve Corporation ★★★★☆ 4 | 330 Reviews

FREE

License	Extension Asset
File size	58.4 MB
Latest version	2.6.1 (sdk 1.13.10)
Latest release date	Aug 4, 2020
Support Unity versions	5.4.6 or higher
Support	Visit site

In Unity, the package manager should open with the SteamVR Plugin selected. Download and import the plug-in from here.

Note: if there is only an 'import' button, that means the asset is already installed on your computer somewhere, you can go ahead and import.


Valve.VR.SteamVR_UnitySettingsWindow

Recommended project settings for SteamVR:

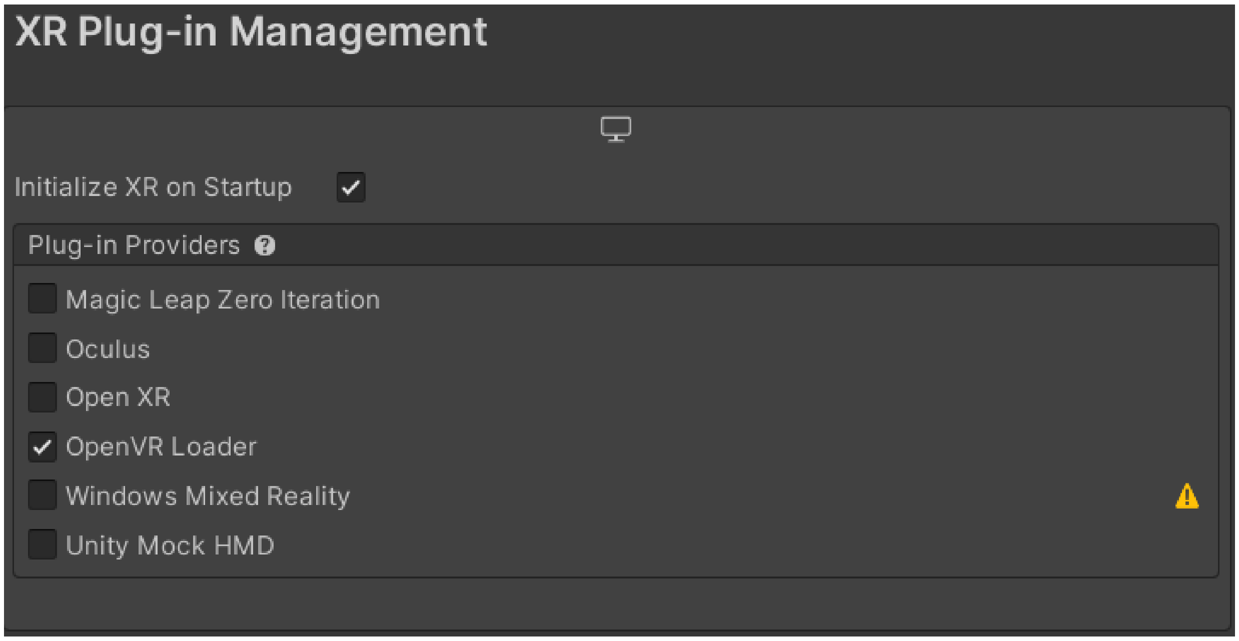
Show Unity Splashscreen (current = True)

After importing, there *might* be a SteamVR popup, as seen in the screenshot, we want to apply all recommended project settings, so click "**Accept All**", as seen in the image. The popup should close itself.

If the popup re-appears later on, do *not* click "Accept All" again, press 'Ignore all' or simply close the popup.

2. Enable OpenVR

In “**Edit > Project Settings**” Find the ‘XR Plugin Management’ tab. Make sure “**OpenVR Loader**” is selected among the XR Plug-in Providers. (*not OpenXR!*) OpenVR is the name of SteamVR’s Unity sdk, and it makes sure Unity is recognizing our headsets correctly.


XR Plug-in Management

Initialize XR on Startup [✓]

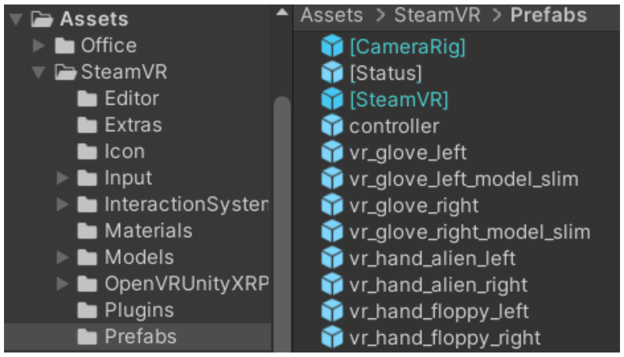
Plug-in Providers

- [ ] Magic Leap Zero Iteration
- [ ] Oculus
- [ ] Open XR
- [✓] OpenVR Loader
- [ ] Windows Mixed Reality
- [ ] Unity Mock HMD

3. Setting up the VR camera, and creating initial controller bindings

After importing the SteamVR plugin, you now have a “SteamVR” folder inside your Unity “Assets” folder.

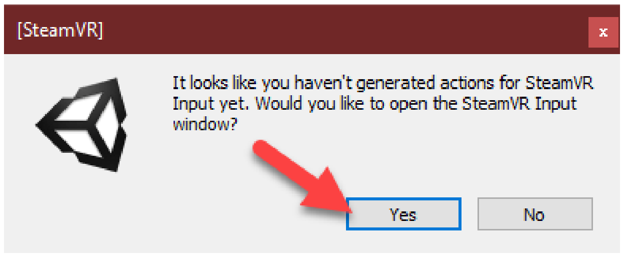
Look in the “Prefabs” folder within the SteamVR folder. You see “SteamVR” and “CameraRig” prefabs:
Drag these two elements into the hierarchy.



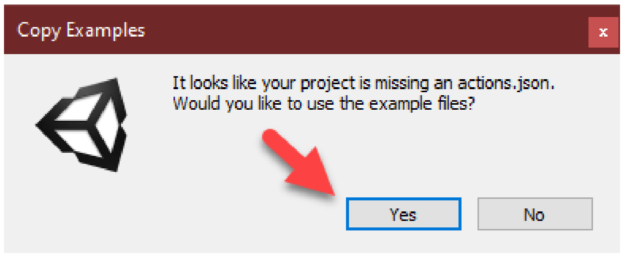
The “Main Camera” that you have in the Hierarchy is now superfluous, and will interfere with the camera inside the CameraRig. So, *deactivate* or *delete* the “Main Camera”.

If you have the VR headset connected to the PC you can already start “Play”.

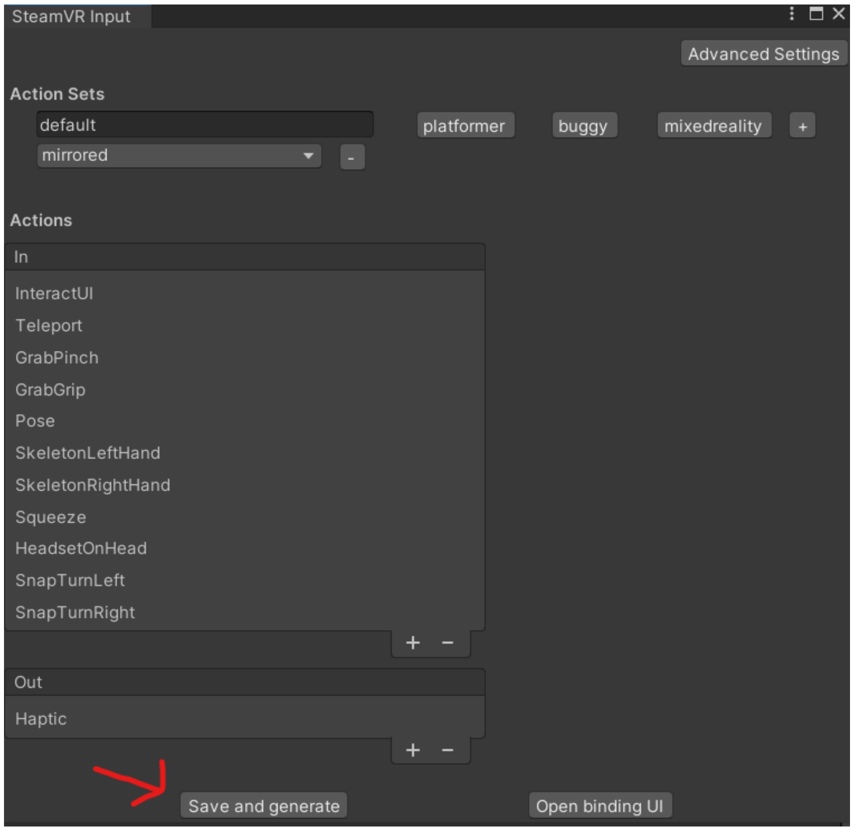
The first time you want to play, SteamVR suggests creating some Input Bindings through the SteamVR Input window. Click“**Yes**”. You then get the hint to “Copy Examples”. Again, click **“Yes”**.


[SteamVR]

It looks like you haven't generated actions for SteamVR Input yet. Would you like to open the SteamVR Input window?


Copy Examples

It looks like your project is missing an actions.json. Would you like to use the example files?



Now you should see the SteamVR Input window. (If not, go to *Window* > *SteamVR Input*.) Here you can see a default set of commonly used VR actions which we are able to edit.

We will come back to it soon, but for now click “**Save and generate**”.

There might now be a popup telling you to **Save** the current Scene. Click “Save”

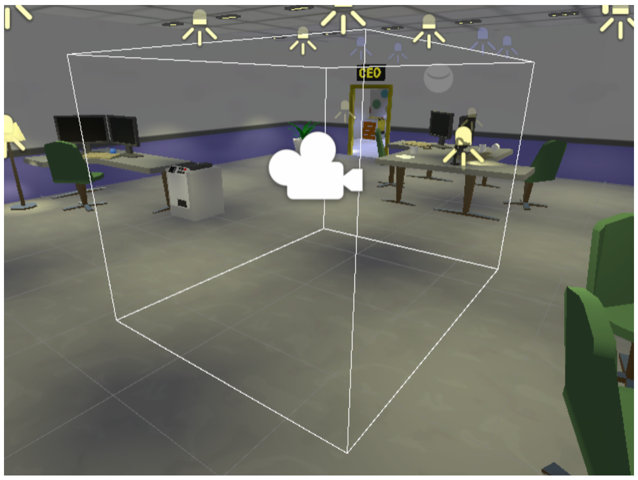
Note: if there are no actions visible at all in the SteamVR input window, something might have gone wrong while importing the SteamVR plugin. Delete the folders ‘SteamVR’, ‘SteamVR_Input’, ‘SteamVR_Resources’ and ‘StreamingAssets’ from your assets folder. Then, head to the package manager and import SteamVR again.

4. First Play

After this configuration, we are ready to “play”. Ensure that your two controllers are turned on, and start play mode in the Unity Editor, with the HMD on your head, controllers in your hands. If all is ok, you should see the office environment, as well as the two moving controllers.

If you are weirdly positioned within the room, you can reposition the CameraRig object with two people: one wearing the HMD, and the “helper” watching the Unity Editor from the connected PC.

Note: When you “play”, Unity might switch to showing the “Game” view. You can switch it back to “Scene” view manually.



While the game is running, in the hierarchy, select the CameraRig (**not** the Camera *inside* the rig) Now watch things when your partner is moving their head: The Camera icon moves around, but the “Rig” stays put.

Now you, the helper, can use the W-key and E-key to move and rotate the CameraRig in the Unity Editor (While the game is still running). Tweak this until, according to your VR partner, this is a nice starting position/rotation.

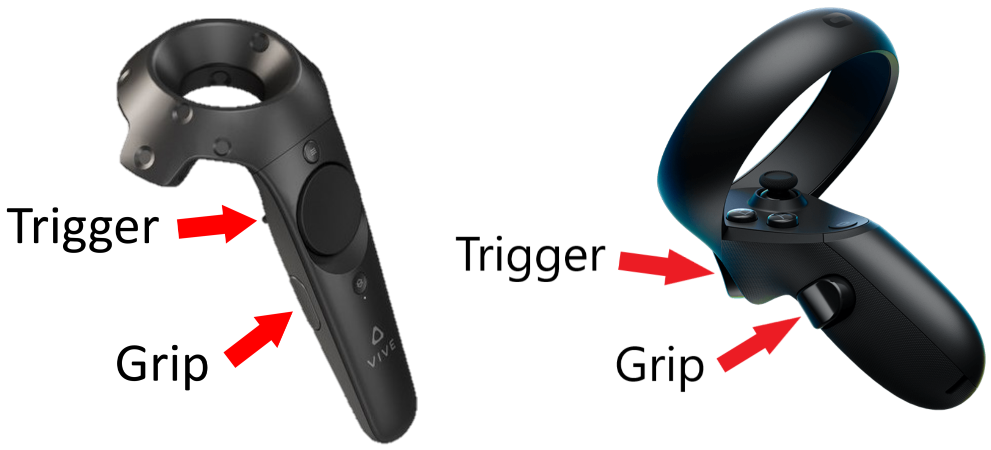
Now comes the trick: look in the Inspector and *write down* the Transform values for **Position** and **Rotation.**

Because as soon as you leave “Play” mode, those values are going to be **reset**. After writing the values down, stop play mode, and enter the Position and Rotation values manually. If you play again, it should look right.

5. Defining Actions and Bindings

For the tutorial, we will have a very basic setup: we want to **grab** and pick up objects in the scene. And we want to “**teleport**” to different locations.

We recommend using the “**Grip**” side button for the “**Grab**” action, and to use the “**Trigger**” for the “**Teleport**” action. However, it is up to you to choose specific buttons for grabbing and teleporting. Now let’s see how to do this with SteamVR.



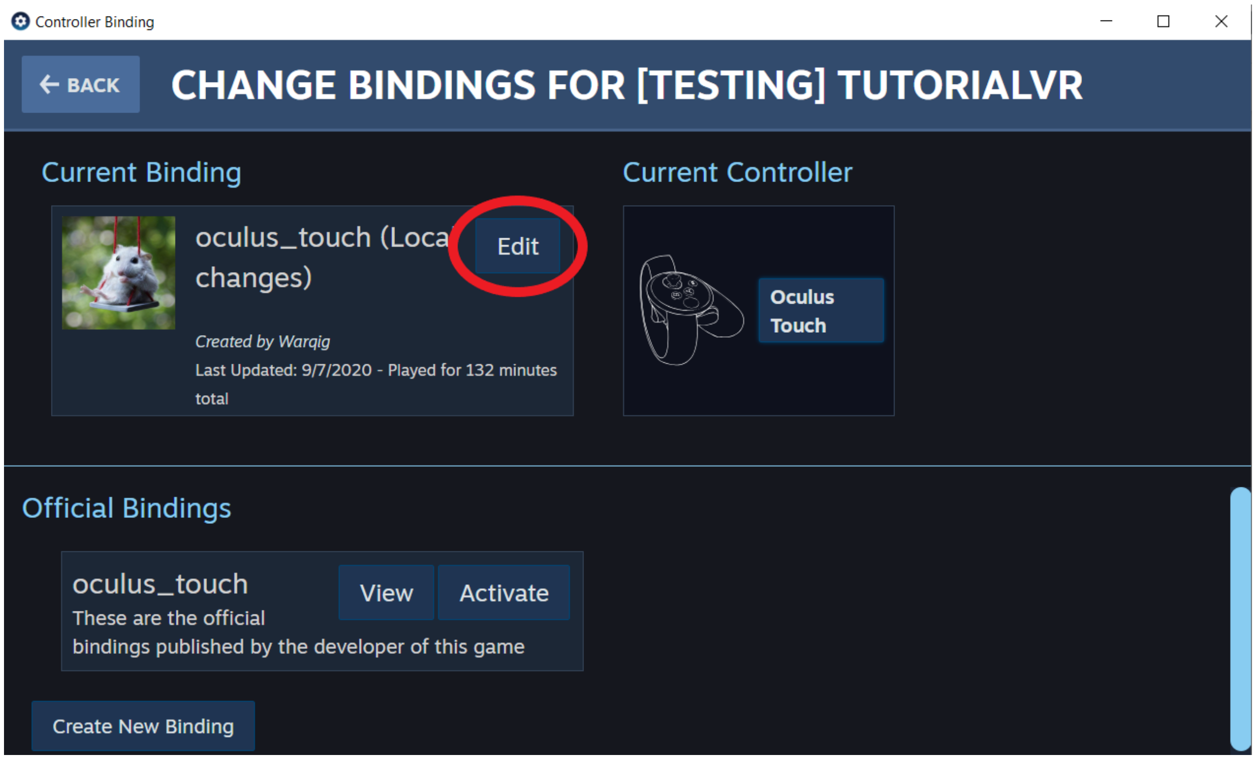
First, use **Window > SteamVR Input** to re-open the SteamVR Input window that we saw before. This time, we want to be more precise in our Actions and Binding.

Step 1: Make sure the ‘Default’ action set is selected, now go to the “Actions” pane. We don’t need all of these example actions. Do the following:

- *Remove* **InteractUI**
- *Remove* **GrabPinch**
- *Remove* **Squeeze**
- *Rename* **GrabGrip** into **Grab**

Step 2: Now use again **Save and generate.** If asked, yes, the “unused input files” can be Deleted.

Step 3: Now we are going to "bind" some of the remaining "Actions" to concrete controller buttons. Ensure that your system is connected to the VR headset, and then click the **Open binding UI.** This opens an interface and it should show your *VR Controller* (In the case of the Oculus Rift and Quest, '*Oculus Touch Controller'*) as a device. It should also show our **Current Binding** (basically the example binding), that we can **Edit.**



oculus_touch (Loca changes)

Created by Warqig

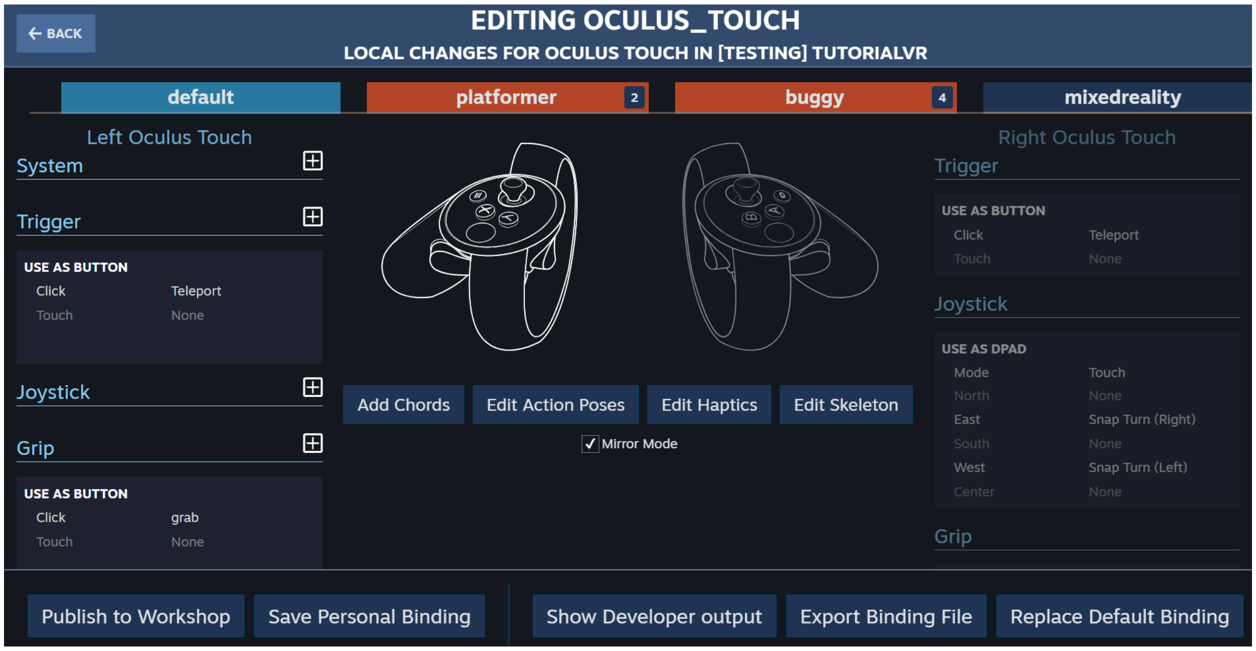
Last Updated: 9/7/2020 - Played for 132 minutes total

oculus_touch

These are the official bindings published by the developer of this game

*Note: If it does not show the Current Binding, you can **activate** the **Official Bindings** shown below the current binding on the screenshot.*

This opens an Editing panel, as shown below (might look slightly different, depending on the headset model used). Note that you can (and should) only edit the **Left** Controller: The **Right** Controller will be identical, since **Mirror Mode** is **enabled**.



Left Oculus Touch

System

Trigger

USE AS BUTTON

Click	Teleport
Touch	None

Joystick

Grip

USE AS BUTTON

Click	grab
Touch	None

✓ Mirror Mode

Right Oculus Touch

Trigger

USE AS BUTTON

Click	Teleport
Touch	None

Joystick

USE AS DPAD

Mode	Touch
North	None
East	Snap Turn (Right)
South	None
West	Snap Turn (Left)
Center	None

Grip

- Make sure **default** is selected at the top and all existing bindings are removed (using the trash icon).
- Click on the Plus symbol for the "**Trigger**". We want to use it as a "Button", so select this option.
- A new field 'button' will appear, this field determines the "**Action**".
- Change the **click** field from "none" to the "**Teleport**" action from our default Action Set.

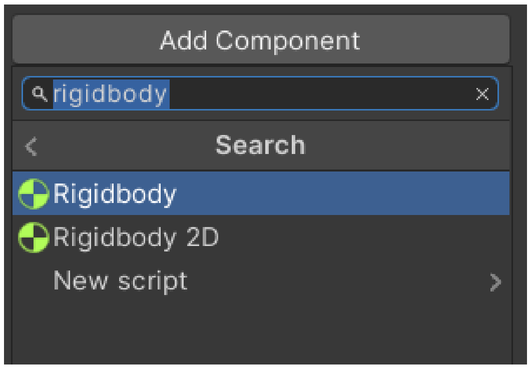
Next, do something similar for the **Grip**, by ensuring that it is configured as a "**Button**", and bind the "**Click**" of the grip to the **grab** action. (We don't use the "Touch"). The final binding should look similar to the picture above. Now we are finished with the "binding"; use the **Replace Default Binding** button near the bottom, and then use the **Save** button.

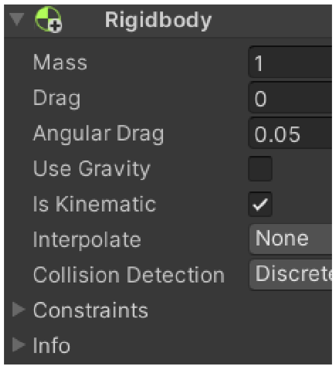
Note: feel free to come back to this overview and edit the bindings to your liking. Make sure to select buttons that feel natural to use for grabbing and teleporting

6. The Controller GameObjects

The two Controllers are represented by the two Controller child objects inside the CameraRig. We want these controllers to have a Unity "**RigidBody**", so they can interact with other GameObjects, like the coffee mugs, chairs, or pretty much everything inside the scene:

- For *both* Controller Gameobjects in the Hierarchy, in the **Inspector** use "**Add Component"**. Search and select **Rigidbody.**
- In the Rigidbody settings, **uncheck** "Use Gravity",
- and **check** "Is Kinematic**"**



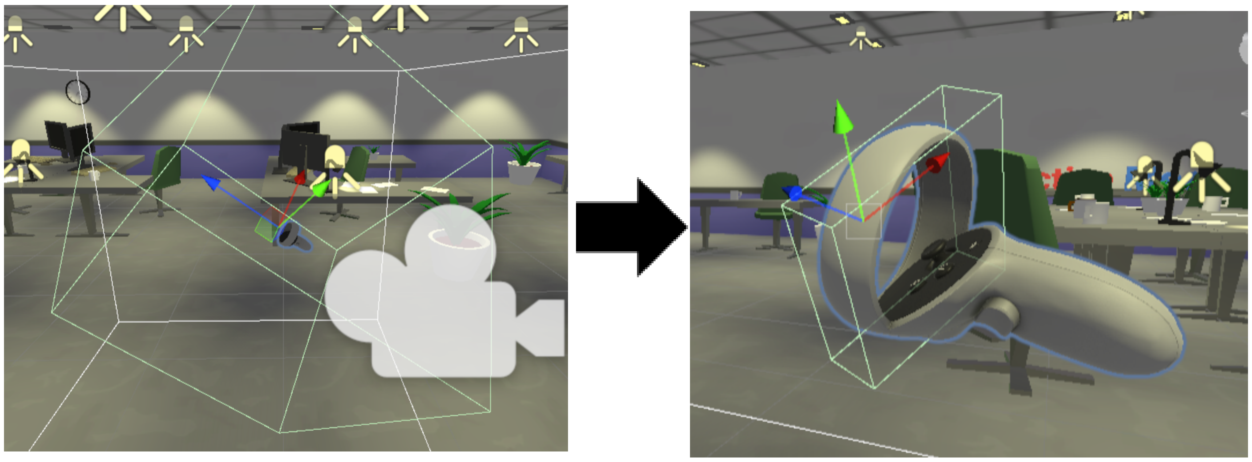

Rigidbody

Mass	1
Drag	0
Angular Drag	0.05
Use Gravity	
Is Kinematic	✓
Interpolate	None
Collision Detection	Discret
Constraints	
Info	

In order to check whether our Rigidbody "touches" other objects, it needs a "**Collider**"; an imaginary shape (a box in our case) that the Unity game engine uses to determine whether it collides with some other object, assuming that the other Gameobject has a similar Collider.

This is simple to do: for the two Controllers, use Add Component again. This time search and add a **Box Collider.**

It is interesting to "Play" again, and watch the Box Colliders in the Scene View (you don't see them in the Game View). As you can see, a box with size **1 X 1 X 1** is far too large. Use the Inspector to set the **Size** of the Colliders to **(X: 0.1, Y: 0.1, Z: 0.03)**. Also, set the **Center** to **(X: -0.01, Y: 0.01, Z: -0.04).** Because every headset model has differently shaped controllers, these settings can vary. Tweak the box collider values until they fit the controllers neatly.



Then, **enable** the **“Is Trigger”** option**.** With this option enabled, the **OnTriggerEnter** method in the script that we are going to attach will be called; if it is *disabled*, you just will “push” other 3D objects away with your controller. (Take care that you change *both* controllers!)

Play again, and have a look at the resized Box Colliders. Thus far, we have set up our controller buttons, but as yet, they don’t *do* anything. This we are going to change in the next section: scripting the controller behaviour.

7. Controller Buttons

Find a place inside your **Assets Folder** where you want to store your scripts. **Right-click**, and choose to **Create a new C# script**. Rename your script to **ActionsTest** and open it in Visual Studio. As usual with scripting in Unity: take care of the precise spelling of script names, including capitalization: one mistake, and Unity won’t recognize your script.

Replace the default script contents with the following code:

```
using UnityEngine;
using Valve.VR;

public class ActionsTest : MonoBehaviour
{
    [SerializeField] SteamVR_Input_Sources handType;
    [SerializeField] SteamVR_Action_Boolean teleportAction;
    [SerializeField] SteamVR_Action_Boolean grabAction;

    void Update()
    {
        if (grabAction.GetState(handType))
        {
        print("Grab: " + handType);
        }

        if (teleportAction.GetStateDown(handType))
        {
        print("Teleport: " + handType);
        }
    }
}
```

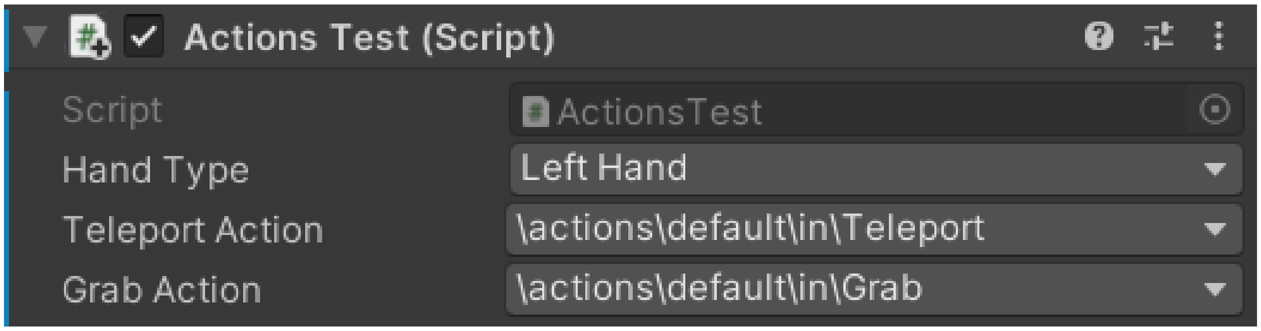
Explanation

This script aims to simply print what button has been pressed. For this we need to import the SteamVR library (line 2) so we can get access to the bindings we have made. After getting references to our controller type and its buttons linked to actions (line 6-8), we have one simple Update() function (which is called every frame).

Within the update function, we check if either of the buttons is being pressed and if so, print for what action and which hand happened to the console.

When you have entered the code, and it compiles correctly, add it to both controllers objects and have a look at the Inspector settings for 'ActionsTest'. You will see our three "SerializedField" variables, and all of them have basically "undefined" values.

Change this: use the dropdown choices, and select **the correct hand type**, as well as the correct **Actions**. Take care that the Controllers each have their own Hand Type!



Script: ActionsTest
Hand Type: Left Hand
Teleport Action: \actions\default\in\Teleport
Grab Action: \actions\default\in\Grab

Now you can "Play", and **watch the Console Window**: check that you see the results of the "print" statements when you use the buttons configured for grabbing and teleporting.

8. Grabbing Objects

Before you proceed with the next script, you should deactivate the “ActionsTest” scripts attached to the two controllers, in the Inspector.

Our next step is to write a script that allows one to “grab” objects, so pick them up, throw them away… We develop this in a number of smaller steps that we can easily test. Our first version is just going to deal with “collisions”. We have set up our trigger colliders above. Now, attach the following ControllerGrabObject script to both Controller objects:

```
using UnityEngine;
using Valve.VR;

public class ControllerGrabObject: MonoBehaviour
{
   [SerializeField] SteamVR_Input_Sources handType;
   [SerializeField] SteamVR_Action_Boolean grabAction;

   private GameObject collidingObject;
   private GameObject objectInHand;

   void SetCollidingObject(Collider col)
   {
       if (!collidingObject && col.GetComponent<Rigidbody>())
       {
           collidingObject = col.gameObject;
           print("CollidingObject");
       }
   }

   public void OnTriggerEnter(Collider other)
   {
       SetCollidingObject(other);
   }

   public void OnTriggerStay(Collider other)
   {
       SetCollidingObject(other);
   }

   public void OnTriggerExit(Collider other)
   {
       if (collidingObject)
       {
           collidingObject = null;
           print("No Colliding Object");
       }
   }
}
```

Explanation

- We start with line 13: “OnTriggerEnter”. This is being called whenever the trigger Collider of the GameObject to which the script is attached enters some “other” Collider. As you can see, that “other” Collider is passed in as a parameter, so we “know” what we have touched so to say.
- Line 5: SetCollidingObject(Collider col) is a simple helper method, that queries “col” for the GameObject to which it is attached, and then stores a reference to that GameObject in the “collidingObject” variable, that we declared on line 4.
- *Summary: we store a reference to the colliding GameObject, only when it has a Rigidbody, and only when we do not already have a non-null reference.*
- Line 18: “OnTriggerStay” will be called repeatedly, until the object exits our trigger collider on the Controller. To be sure we do track the “other” GameObject, we call SetCollidingObject again. Normally speaking this should have no effect (why?), but for “stability” it seems better to call it anyway, so we won’t “miss” a collision.
- Line 22: “OnTriggerExit”. Now we no longer have a collision, and, if we had set a non-null collidingObject, we now reset it to “null”. For our first test, we also “print” this when it happens.

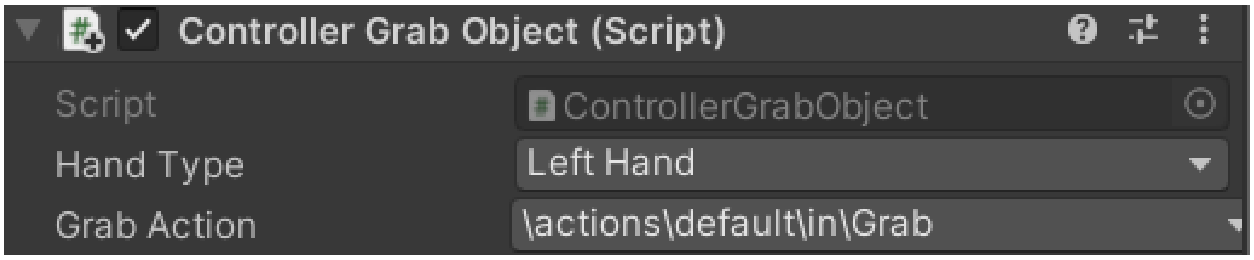
Next, add the following three methods to the script, in order to test our grabbing functionality;

```
40      void GrabObject()
41      {
42          objectInHand = collidingObject;
43          print("Grab Object");
44      }
45
46      void ReleaseObject()
47      {
48          objectInHand = null;
49          print("Release Object");
50      }
51
52      void Update()
53      {
54          if (grabAction.GetLastStateDown(handType))
55          {
56              if (collidingObject)
57              {
58                  GrabObject();
59              }
60          }
61          if (grabAction.GetLastStateUp(handType))
62          {
63              if (objectInHand)
64              {
65                  ReleaseObject();
66              }
67          }
68      }
```

Explanation

- We have added a "GrabObject" (line 35) and a "ReleaseObject" (line 41) method, inside the *ControllerGrabObject* script
- Find out how they keep track of a "collidingObject" and an "objectInHand". (We suppose here that when we "grab" something, we hold it in our hand)
- These two new methods are being called from the "Update" method (line 49).
- Inside "Update", we use "grabAction". This time we use a method called "GetLastStateDown", where we must specify the handType. It returns "true" when the button is now "down", but it was "up" in the previous state. So, it signals the moment when you "press" the button.
- If at that moment of "pressing" the button, we actually are colliding with something, we call "GrabObject".
- Similarly, when we have that "GetLastStateUp", and we have something in our hand, we call "ReleaseObject".

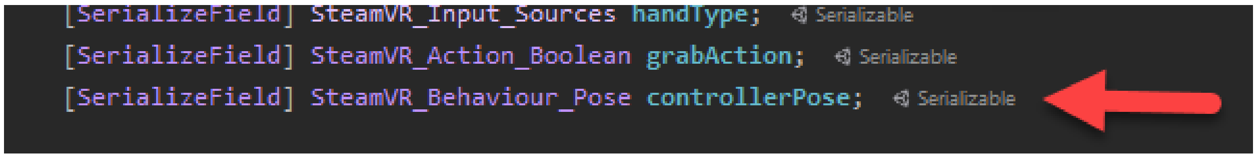
We use the “grabAction'', similar to how we used it before in our “ActionTest” script. That also means that we have to use the Inspector for the Controllers, and fill these fields in the inspector with the correct values:


Controller Grab Object (Script)

Script ControllerGrabObject
Hand Type Left Hand
Grab Action \actions\default\in\Grab

Go ahead and **test if your code is working** by trying to grab objects and looking if ‘Grab object’ and ‘Release object’ are being printed to the console when doing so. We just implemented print statements, so don’t worry if you are not able to pick up objects yet.

The final step is to implement the “Grab” and “Release”. Before we can do this, a slight modification has to be made to our script; add an extra “controllerPose” field. It corresponds to the “Pose” Action, that keeps track of the “pose” of a Controller, that is, its position and rotation in the real world.


```
    [SerializeField] SteamVR_Input_Sources handType;
    [SerializeField] SteamVR_Action_Boolean grabAction;
    [SerializeField] SteamVR_Behaviour_Pose controllerPose;
```

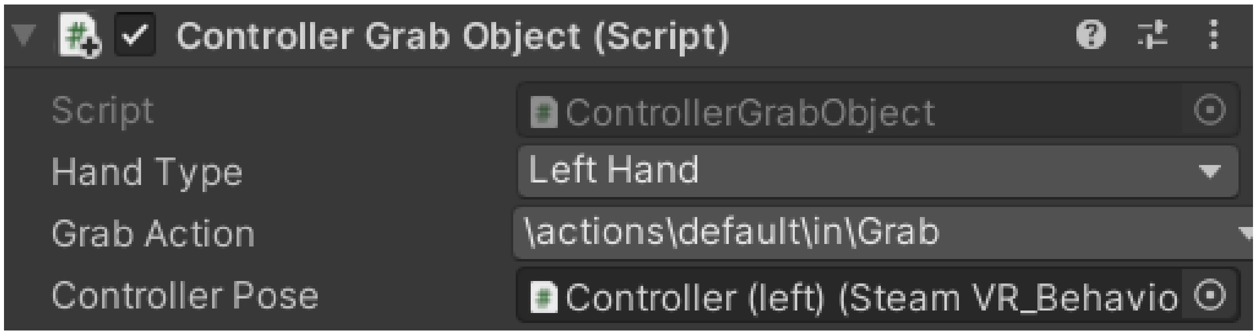
Next, replace the “GrabObject” and “ReleaseObject” by the following new methods, and add an additional method called “AddFixedJoint”;

```
    void GrabObject()
    {
        objectInHand = collidingObject;
        collidingObject = null;
        var joint = AddFixedJoint();
        joint.connectedBody = objectInHand.GetComponent<Rigidbody>();
    }

    void ReleaseObject()
    {
        if(GetComponent<FixedJoint>())
        {
            GetComponent<FixedJoint>().connectedBody = null;
            Destroy(GetComponent<FixedJoint>());
            Rigidbody rb = objectInHand.GetComponent<Rigidbody>();
            rb.velocity = controllerPose.GetVelocity();
            rb.angularVelocity = controllerPose.GetAngularVelocity();
        }
        objectInHand = null;
    }

    FixedJoint AddFixedJoint()
    {
        FixedJoint fx = gameObject.AddComponent<FixedJoint>();
        fx.breakForce = 20000;
        fx.breakTorque = 20000;
        return fx;
    }
```

Explanation

- A “FixedJoint” is a Component that can “link” two GameObjects together. It is a “physical” joint, so it can “break” when you apply enough force or enough torque. (“torque” is like “twisting force”) here, we simply specify large values for these forces and torques, because we don’t want it to happen.
- The “FixedJoint'' is created in GrabObject. Then it is linked to the Rigidbody of the Gameobject that we “have in our hand”. From now on, the two GameObjects will move together.
- ReleaseObject is simple: provided we still have that FixedJoint object (i.e., when it is non-null), we “unlink”, and we destroy the FixedJoint.
- But we add one refinement: At the moment of the “release”, we can retrieve the current velocity and angular velocity from the controller “Pose” action. We use that to give the object we are releasing that velocity and angular velocity. If we didn’t it would “drop dead” on the ground. Instead, it can now be thrown away.

Like before, we have three "Serialized Field" variables, and you must set their values inside the Inspector for **both** controllers:


Controller Grab Object (Script)
Script: ControllerGrabObject
Hand Type: Left Hand
Grab Action: \actions\default\in\Grab
Controller Pose: Controller (left) (Steam VR_Behavio

Now you can **Play**, and see whether you can grab and release objects.

9. Teleporting

Finally, we want to be able to "teleport", using a laser beam, that we activate with the trigger button. If you remember: we bound that button to the "Teleport" action, and we are going to use that. First, however, we need a simple GameObject that represents a "laser beam". This is easy:

- Create a new **Cube** in the Scene, name it 'laser'.
- Remove the attached Box Collider.
- Set the **Position** to **(0,0,0)**
- Set the **Scale** to **(X:0.005, Y: 0.005, Z:1)**
- You now have a long "beam" object.
- Next, create a new Material somewhere in your asset folder, and call it "LaserMat".
- Use the Inspector to set it to any colour of your choosing.
- Drag the Material onto the "laser" beam, it should turn to the selected colour.
- Finally, we turn our "laser" into a Prefab: drag it into the "TeleportReticle" folder
- Delete the original object inside the Scene.

There is already some other Prefab in that folder, called "ReticlePrefab". We will use both Prefabs in the new script "LaserPointer" below.

```
1   using UnityEngine;
2   using Valve.VR;
3
4   public class LaserPointer : MonoBehaviour
5   {
6       [SerializeField] SteamVR_Input_Sources handType;
7       [SerializeField] SteamVR_Behaviour_Pose controllerPose;
8       [SerializeField] SteamVR_Action_Boolean teleportAction;
9
10      // Laser variables
11      [SerializeField] GameObject laserPrefab;
12      private GameObject laser;
13      private Vector3 hitPoint;
14
15      // Reticle variables
16      [SerializeField] Transform cameraRigTransform;
17      [SerializeField] GameObject teleportReticlePrefab;
18      private GameObject reticle;
19      [SerializeField] Transform headTransform;
20      [SerializeField] Vector3 teleportReticleOffset;
21      [SerializeField] LayerMask teleportMask;
22      private bool shouldTeleport;
23
24      private void Start()
25      {
26          laser = Instantiate(laserPrefab);
27          reticle = Instantiate(teleportReticlePrefab);
28      }
29  }
```

Explanation
This is the start of the 'LaserPointer' script, which is still mostly a bunch of variables that have not been initialized. Because we want to see the laser only when pressing and holding the teleporting button, both the *laser* as well as the *reticle* should be instantiated ('spawned in') while unity is running. To do this we need a reference to the objects that we are going to instantiate, as seen on lines 11 and 17. In the start method, which runs the first frame Unity is running, we then instantiate these prefabs into GameObjects and store those in private variables, which we can use later on in the script, lines 26 and 27.

Now we can start adding functions to our script. Note that the teleporting will still not function, as the new methods are not yet called anywhere.

```
    private void ShowLaser(RaycastHit hit)
    {
        laser.SetActive(true);
        laser.transform.position = Vector3.Lerp(controllerPose.transform.position, hitPoint,0.5f);
        laser.transform.LookAt(hitPoint);
        Vector3 laserScale = laser.transform.localScale;
        laser.transform.localScale = new Vector3(laserScale.x, laserScale.y, hit.distance);
    }

    private void Teleport()
    {
        shouldTeleport = false;
        reticle.SetActive(false);
        Vector3 difference = cameraRigTransform.position - headTransform.position;
        difference.y = 0;
        cameraRigTransform.position = hitPoint + difference;
    }
```

Explanation
The *ShowLaser* method will be called every frame when holding the teleport button down. The instantiated laser object will then be positioned and scaled so that the beam is visible between the hand of the user and the surface they are aiming at (‘hitPoint’).

The Teleport() method will be called when the player is releasing the teleport button, for actually moving the player (cameraRigTransform) to the new position. After teleporting, the teleport reticle is disabled again.

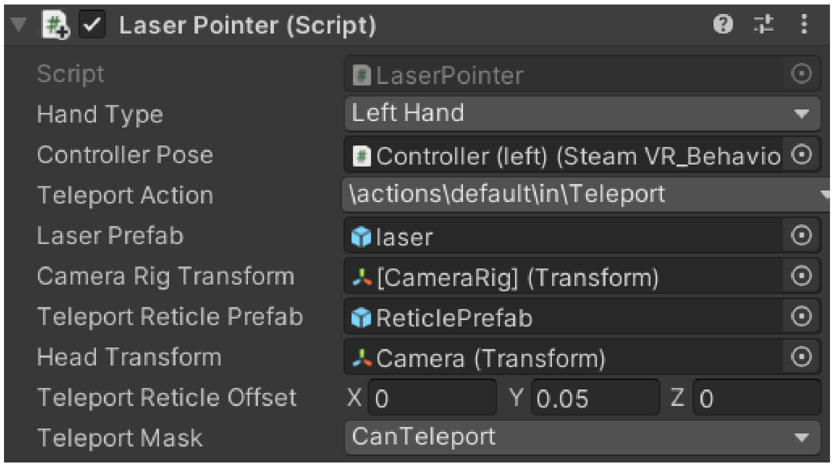
Finally, add the following code to your *LaserPointer* class; it will detect if our teleport action is being pressed or released, and teleport the player to the location they are aiming at.

```
private void Update()
{
    if (teleportAction.GetState(handType))
    {
        RaycastHit hit;
        Vector3 controllerPos = controllerPose.transform.position;
        if (Physics.Raycast(controllerPos, transform.forward, out hit, 100))
        {
            hitPoint = hit.point;
            ShowLaser(hit);
            if (((1<<hit.collider.gameObject.layer) & teleportMask) != 0)
            {
                print( "Teleport hit!" );
                reticle.SetActive(true);
                reticle.transform.position = hitPoint + teleportReticleOffset;
                shouldTeleport = true;
            } else {
                reticle.SetActive(false);
                shouldTeleport = false;
            }
        }
        else
        {
            laser.SetActive(false);
            reticle.SetActive(false);
        }
    }
    else
    {
        laser.SetActive(false);
        reticle.SetActive(false);
    }

    if (teleportAction.GetStateUp(handType) && shouldTeleport)
    {
        Teleport();
    }
}
```

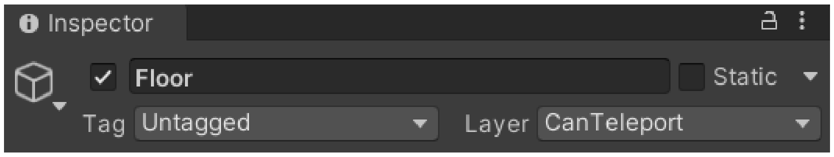
Explanation
Here comes the most important part. Update() will be called every frame while unity is running, you should already know this. In the Update method, we check if the button we bound to the teleporting action (probably 'Trigger') is being held down, using teleportAction.GetState() on line 50. When it is held down, we shoot a *raycast* from the tip of the user's controller forward. A raycast is an invisible ray that can detect which object it collides with. We use this extremely useful tool to check if the player is aiming at the floor, by comparing the layer of the colliding object to a predefined 'layermask'. This is because we don't want the player to teleport onto all the different objects in the scene, but only the floor. When the raycast does not hit anything (*else*, line 64), we disable the reticle (the 'target' visual) as it should not be visible when the player is not aiming at the floor. When the raycast is not colliding with anything (*else*, line 69), we also disable the laser.
Finally, we call our *Teleport()* method, when the teleporting button is released, line 83

Add the script to both Controller Objects, and as usual, use the Inspector to fill the "SerializeField" values, see screenshot.

Note: The last value is a "Layer Mask". This layer type is already added to the layers in our Unity project. We use it to "tag" the objects that can be used as a target for the laser beam, so every surface that should be available for teleporting.


Laser Pointer (Script)

Script	LaserPointer
Hand Type	Left Hand
Controller Pose	Controller (left) (Steam VR_Behavio
Teleport Action	\actions\default\in\Teleport
Laser Prefab	laser
Camera Rig Transform	[CameraRig] (Transform)
Teleport Reticle Prefab	ReticlePrefab
Head Transform	Camera (Transform)
Teleport Reticle Offset	X 0 Y 0.05 Z 0
Teleport Mask	CanTeleport


Inspector

Floor Static

Tag Untagged Layer CanTeleport

Select the floor in the scene hierarchy, and set its 'layer' dropdown to *CanTeleport*.

Done!

If you are able to grab objects and teleport around, you are done with the tutorial! Go ahead and try interacting with the environment. There are some easter eggs to explore :)

[Optional] Apply what you learned

For this final exercise you will need to apply what you just learned by adding additional interaction to the scene: make it so that you **shoot objects from your hands** on a button press.

You can achieve this by following these steps:

- Add an action to the action set in the SteamVR Input window
- Bind this action to an unused button in the binding UI
- Add a new script to both controllers
- Test if your binding works by printing to the console, you can check if the button is pressed inside the Update() method.
- Import a model from the Asset store, or one of your own Maya/Blender models, and make a prefab out of it, Make sure to attach a collider and rigidbody.
- Have your script instantiate this new prefab when your button press is detected (*Look at the documentation for ways to give the instantiated prefab the right position and rotation)*
- Add a force to this prefab to make it 'shoot out of your hand'. You can use *GetComponent* to retrieve the Rigidbody from the instantiated prefab, and use *AddForce* on the rigidbody of the instantiated prefab to give it velocity.

Note: the AddForce() method expects a Vector3 as an argument, as a 'force' is directional. You can use the ControllerPose.transform.forward vector to get the direction your controller is pointing. Make sure to multiply this vector with a large number so the force is significantly high.

Go through all previous steps of this assignment if you need a reminder of how to do things.