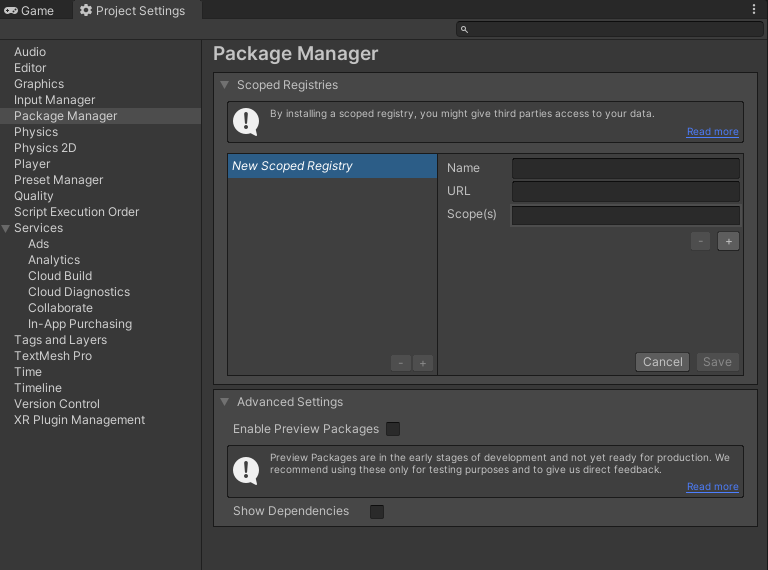
Tutorial – Setting up VR for Oculus Quest 2 in Unity

The goal for this tutorial series is ensure you have the knowledge to set up an Oculus device to start developing for VR. By follow the next few steps you will be at a stage to start active development.

1. Create a new project using **URP** for **Unity** via Unity Hub.
2. Open the **Edit** > **Project Setting** > **Package Manager** > **Advance Settings** and tick **Enable Preview Packages**.  
   A *Package Manager* prompt will appear, choose **I Understand**.
3. Open **Window** > **Package Manager** and open the drop down at the top, it should say **Packages: In Project** and choose **Unity Registry**.
4. From here you will need to install the following:

* **Oculus XR Plugin**, this will make oculus integration easier.
* **XR Plugin Management**, this should be installed automatically with the one above.
* **XR Interaction Toolkit**, after this is installed, it will restart Unity.

1. Graphical user interface, text

   Description automatically generatedGo to the **XR Interaction Toolkit** and expand the Samples, you will be importing both
2. From here go to Unity’s **Assets** > **Samples** > **XR Interaction Toolkit** > **0.10.0-preview.7** > **Default Input Actions**
3. Open the **XRI Default Left Controller** and in the **Inspector** you see a variety of options you can ignore for now; at the top you will need click the **Add to ActionBasedController default** button.
4. Do the same thing for **XRI Default Right Controller**.
5. Now when you open the **XRI Default Input Actions** map you will see options to map the interactions of the two controllers and the headset.
6. Graphical user interface, application

   Description automatically generatedOnce that is done go to **Edit** > **Project Settings** > **Preset Manager** and fill it in like the image below.
7. Graphical user interface, application

   Description automatically generatedNow go to **Edit** > **Preferences** > **External Tools** and in the **Android** section change the defaults of the **JDK**, **SDK** and **NDK** to the following:
8. Graphical user interface, text, application

   Description automatically generatedOnce that is done go to **Edit** > **Project Setting** > **XR Plug-In Management** > **Android** and tick the Oculus option. Make sure oculus isn’t ticked under any other setting otherwise it will cause build failures later.
9. Graphical user interface, text, application

   Description automatically generatedGo to **Edit** > **Project Setting** > **Player** and change the **Company Name** and a **Product Name** to an appropriate name for the application you are developing.
10. Graphical user interface, text

    Description automatically generatedWhile in **Player** go to **Other Settings** > **Identification** > **Minimum API Level** and change it to **Android 6.0 ‘Marshmallow’ (API level 23)**
11. Finally go to **File** > **Build Settings** and change the platform to android by selecting **Android** and pressing the **Switch Platform** button at the bottom of the window.  
    Graphical user interface

    Description automatically generated
12. After a moment of processing your application should be ready for VR development! Its not over yet however, after finishing the setup to start creating for VR we now need to set up a Unity scene to function!

Tutorial – Oculus Quest 2 Scene Setup

1. Inside the Sample Scene provided by Unity for the URP showcase you are going to go to **GameObject** > **XR** > **Room Scale XR Rig (Action-Based)**, this will create an **XR Rig** in your **Hierarchy**.Graphical user interface

   Description automatically generated
2. A screenshot of a computer

   Description automatically generated with medium confidencePlace the XR Rig in the scene so the bottom of the cube it creates is close to the floor of the scene:

1. A picture containing text, computer

   Description automatically generatedThen select **GameObject** > **XR** > **Interaction Manager** and add that to the scene
2. Find the **URP\_Oculus\_Quest2\_Assets.unitypackage** on canvas or from your teacher and open it into your project, importing all assets provided.
3. Graphical user interface, text, application

   Description automatically generatedFrom there go to the **Hierarchy**, open the **XR Rig** > **Camera Offset** > **LeftHand Controller**, go into the bottom of the **XR Controller** Script and under **Model** you will set the **Model Prefab** to *Q2LeftControllerPrefab* and set **Model Transform** to LeftHand Controller.
4. Graphical user interface, text, application

   Description automatically generatedFollow the same process above for the RightHand Controller using the right equivalents.
5. Graphical user interface, application

   Description automatically generatedGo to the XR Interaction Manager and add the **Input Action Manager (Script)**; in **Action Assets** set it to one (1) and add the *XRI Default Input Actions* asset.
6. From this point if you **build** and run you should be able to successfully stand in a scene and have controllers appear.

Tutorial – Oculus Quest 2 – Teleportation

In this tutorial, we are going to start implementing the Locomotion System from the XR Interaction Toolkit, this will provide movement for the XR Rig. We will look at how the Locomotion System works with the XR Rig, implement teleportation for area teleporting or anchor teleporting and configure snap turns.

**Teleportation Area**: like the name suggests, the surface of an area designated by the teleportation script becomes accessible to the user and gives them the ability to move freely in a specific area and teleport anywhere in that area without the usual discomfort of walking in VR.

**Teleportation Anchor**: working in a similar way to the area, we can choose a surface to be teleportable, but unlike the area the use becomes stuck on a specified point while on the area, so while great for designating play areas it isn’t ideal if you want to give the player freedom to roam larger areas then the guardian zone.

With this knowledge, let’s get started!

1. First, lets start by creating an empty game object in the Hierarchy, we’ll call it *Locomotion System*. On this object we are going to add a few scripts:
   * **Locomotion System**: This allows the system to handle the movement of an XR Rig
   * **Teleportation Provider**: This will handle teleportation movement in the Locomotion System
   * Graphical user interface

     Description automatically generated**Snap Turn Provider (Action-based)**: This will allow use to turn or look behind us instantly using the controller’s joystick.
2. From this point we need to assign the values needed in these scripts
   * **Locomotion System**: You need to set the XR Rig to the XR Rig in the Hierarchy.
   * **Teleportation Provider**: Set the Locomotion System in the script to itself in the Hierarchy.
   * Graphical user interface, text, application

     Description automatically generated**Snap Turn Provider (Action-based)**: Set the Locomotion System in the script to itself in the Hierarchy. Set the Left Hand Snap Turn Action’s Use Reference to true. Look into the Input Action Reference and set it to XRI LeftHand/Turn.  
     Follow the same procedure for the Right Hand Snap Turn Action.
3. Next from the Hierarchy go into the Example Assets > Workshop Set > Ground and add a Teleportation Area Script. This will allow us to test the teleportation functionality. Make sure to set the interaction manager to the one in the Hierarchy and set the Custom Reticle to the Marker prefab.  
   Graphical user interface, text, application

   Description automatically generated
4. Text

   Description automatically generatedNow we need to refactor the controller, the reason for this is current the default controllers work but they only have a few options for buttons that we can actively call, this means that we may want to use the same button for teleporting as the one used for game play. What we are going to go is create a script that switches between controller states depending on the button press to quickly implement teleportation. The image on the left is how the hierarchy look at this point. After the next few steps it should look like the one on the right.  
    Text

   Description automatically generated

1. First, we are going to rename **LeftHand Controller** to **LeftHand Gameplay Controller** and do the same for the **RightHand Controller**. For each of these Gameplay Controllers we are going to create an Empty child and call it **LeftHand Transform** for Left and the same for the Right. Make sure the transform is reset. At this point it should look like the image below:  
   Graphical user interface, text

   Description automatically generated
2. From here we are going to create two Empty children of the **Camera Offset**, one will be called **LeftHand** and the other **RightHand**. From here we will attach the **LeftHand Gameplay Controller** to the **LeftHand** and follow the same process for the **RightHand**. At this point it should look like the image below:  
   Text

   Description automatically generated
3. Now we are going to want to check the **LeftHand Gameplay Controller**’s Inspector for the **XR Controller (Action-Based)** script. At the bottom of the script there should be two empty sockets under the Model heading as shown below:  
   Graphical user interface, text, application

   Description automatically generated  
   In the **Model Transform** we will attach the child of this gasmeobject, the **LeftHand** **Transform**. For the **Model Prefab** we will go into the Prefabs folder in our Assets and grab the **Q2LeftControllerPrefab**. After this it should look like the image below  
   Graphical user interface, application

   Description automatically generated  
   Follow these same steps for the **RightHand** and it should look like the image below.  
   Graphical user interface, text, application

   Description automatically generated
4. Next, we will duplicate the **LeftHand Gameplay Controller** and call it the **LeftHand Teleport Controller**. Make sure they are both still a child of the **LeftHand**.
   1. Now in the **LeftHand Gameplay Controller**’s Inspector we are going to remove all components except the XR Controller
   2. We are then going to add an **XR Direct Interactor** and attach the **XR Interaction Manager** to the **Interactions Manager**
   3. Then you will need to add a trigger to the controller, I recommend using a **Sphere Collider** that has been set to a **Trigger**. This will allow us to interact with objects in the scene by grabbing them if they are in the trigger’s volume, without this the **XR Direct Interactor** will not work.
   4. You should also make the **Sphere Collider**’s radius smaller depending in the game, I recommend setting the radius to 0.2f.
   5. Graphical user interface, text, application

      Description automatically generatedOnce that is done the **Inspector** should look like the following:

Follow these same steps for the **RightHand Gameplay Controller**.

1. Next we will need to create a script that switches between the Gameplay Controller and the Teleport Controller for each hand. Lets create a new script in our Scripts folder called **TeleportActivate** and open it in Visual Studio. We are going to remove Using.System lines and replace them with UnityEngine.InputSystem and UnityEngine.Events. We can also remove the update as we will not require this function.   
   From here you should see something like this:

using UnityEngine;

using UnityEngine.InputSystem;

using UnityEngine.Events;

public class TeleportActivate : MonoBehaviour

{

// Access the GameObject that contains the teleport controller script.

public GameObject teleportController;

// Reference to the Input Action Reference that contains the button mapping

data for activation.

public InputActionReference teleportActivateReference;

// Start is called before the first frame update

void Start()

{

}

}

using UnityEngine;

using UnityEngine.InputSystem;

using UnityEngine.Events;

public class TeleportActivate : MonoBehaviour

{

// Start is called before the first frame update

void Start()

{

}

}

1. Now we need to declare some public variables to access other scripts we need to be able to access the Teleport Controller and the Input Action button mapping.
2. We then want to create some unity event calls that will occur at when we either turn on the teleport controller or turn off the controller, this will let us switch rapidly between the control schemes and functionality.

using UnityEngine;

using UnityEngine.InputSystem;

using UnityEngine.Events;

public class TeleportActivate : MonoBehaviour

{

// Access the GameObject that contains the teleport controller script.

public GameObject teleportController;

// Reference to the Input Action Reference that contains the button mapping

data for activation.

public InputActionReference teleportActivateReference;

[Space]

[Header("Teleport Events")]

// These will group Unity event calls that you can add to in the inspector

public UnityEvent onTeleportActivate;

public UnityEvent onTeleportCancel;

// Start is called before the first frame update

void Start()

{

}

}

1. Once that is added we will need to create come code to invoke to setup the controllers and allow them to switch.

// This will let us call a series of events created in the onTeleportActivate

events in the inspector

private void TeleportModeActivate(InputAction.CallbackContext obj) =>

onTeleportActivate.Invoke();

// This will delay the call of the DelayTeleportation function for 0.1 of a

second

private void TeleportModeCancel(InputAction.CallbackContext obj) =>

Invoke("DelayTeleportation ", .1f);

// This will let us call a series of events created in the onTeleportCancel

events in the inspector

private void DelayTeleportation () => onTeleportCancel.Invoke();

1. Finally in the Start() we will need to add some calls to have the game switch modes based on button presses

using UnityEngine;

using UnityEngine.InputSystem;

using UnityEngine.Events;

public class TeleportActivate : MonoBehaviour

{

// Access the GameObject that contains the teleport controller script.

public GameObject teleportController;

// Reference to the Input Action Reference that contains the button mapping

data for activation.

public InputActionReference teleportActivateReference;

[Space]

[Header("Teleport Events")]

// These will group Unity event calls that you can add to in the inspector

public UnityEvent onTeleportActivate;

public UnityEvent onTeleportCancel;

// Start is called before the first frame update

void Start()

{

// An Interaction with the teleportActivationReference has been completed

and performs a callback to the TeleportModeActivate.

teleportActivateReference.action.performed += TeleportModeActivate;

// An Interaction with the teleportActivationReference has been cancelled

and performs a callback to the TeleportModeCancel.

teleportActivateReference.action.canceled += TeleportModeCancel;

}

// This will let us call a series of events created in the onTeleportActivate

events in the inspector

private void TeleportModeActivate(InputAction.CallbackContext obj) =>

onTeleportActivate.Invoke();

// This will delay the call of the DelayTeleportation function for 0.1 of a

second

private void TeleportModeCancel(InputAction.CallbackContext obj) =>

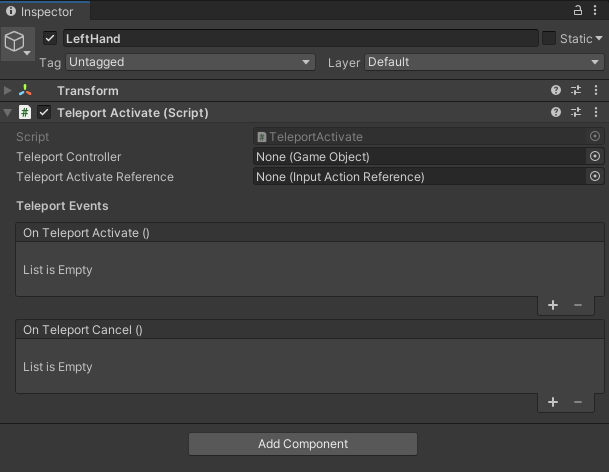
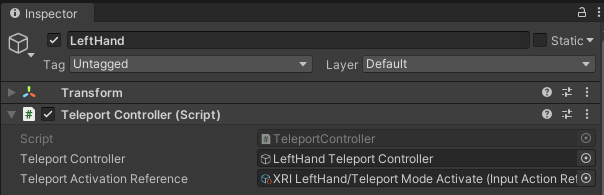
Invoke("DelayTeleportation", .1f);

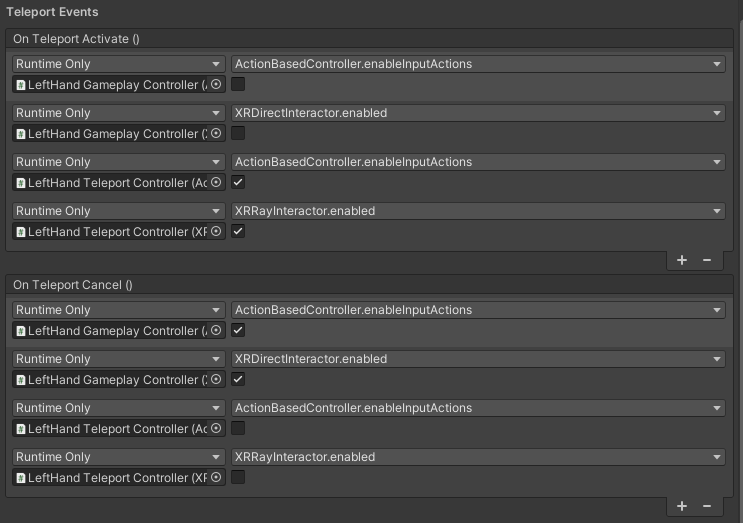
// This will let us call a series of events created in the onTeleportCancel

events in the inspector

private void DelayTeleportation () => onTeleportCancel.Invoke();

}

1. Next, we need to go back to the **LeftHand** and **RightHand** components and set them up. We’ll start with the left and you can follow the same steps for the right
2. First we will attach the **LeftHand Teleport Controller** gameobject and then look through the **Input Action References** to find **XRI LeftHand/Teleport Mode Activate**.

1. Next we need to hit the + and add 4 OnTeleportActive() calls and 4 OnTeleportCancel() calls as shown below:  
   This will allow the correct components to be turned off and on for controller switching
2. From here you will need to select LeftHand Teleport Controllers for and turn off the **Enable Input Actions** in the **XR Controllers** and turn off the **XR Ray Interactor** component.  
     
   In the **XR Controller > Select Action**, set the value for these teleport controllers to **XRI LeftHand/Teleport Select**.
3. Now you just need to follow steps 14 to 17 but for the RightHand.
4. From here you can add into the Hierarchy, XR > Teleportation Anchors and use them by rotating the scene to meet the angle you want the player to face, you can also at a marker to tell a player that they can teleport to the surface or indicate the type of teleportation it will be. But from here you should have a working simple Teleportation system for you to use!

Tutorial / Exercise – Name of session here

This is just some example jargon junk from an old tutorial, used to show layout etc

We will then create our own shader to make use of the lights, replacing Gamebryo’s default pipeline.

We will be using a scene called **TheRoom.nif** that is hosted on E-Learn as **TheRoom.zip**. It is a sizeable download so check that your teacher does not already have it on the campus network.

You may want to start with a clean Application.h/.cpp such as the one located in **RenderInputApp.zip**, or continue using your current one. It just needs to be able to load the nif and display it. The nif **does not contain a camera**, so will need to use your own. It is suggested you start the camera translated to **(0,-250,250)** and that you set its **near** and **far** distance to **[1,10000].**

Set your application to load and display the scene. You will see that it is black, because it does not contain any lights, which we will now add. **Don’t apply your own shader to the scene!**

Gamebryo supports 4 different light classes:

**NiAmbientLight** – Ambient lighting that affects all surfaces regardless of facing.  
 **NiDirectionalLight** – Direction lights that reach forever in a set direction from all points.  
 **NiPointLight** – A light that emits from a single point for a certain radius.  
 **NiSpotLight** – A light that emits from a single point in a set direction with a specified cone  
 and distance.

Adding an NiAmbientLight:

1. In OnCreate() we will add an **NiAmbientLight** and have it effect the entire scene.
2. For lights to work we must specify an **NiAVObject** for it to effect. If the object is a node in a scene graph then the light will affect all child nodes and their children.
3. In OnCreate() create a light in the following way, attaching it to the root of the scene graph so it affects everything:

// this is a code sample

// it uses consolas font with no paragraph spacing

// and has a solid border

int main( int argc, char\* argv[] )

{

return 0;

}

We keep making more stuff.

Until we’re done!

Use boxes like this to explain samples or small snippets when it makes sense to do so!

// this is a code sample

// it uses consolas font

int main( int argc, char\* argv[] )

{

return 0;

}

More still!