XRInput Command System (XRCS) Documentation

The XRInput Command System (XRCS) is not necessary to process XR inputs using Unity 2019.2's XRInput system. However it makes it so that the developer will not need to write a single line of XRInput code.

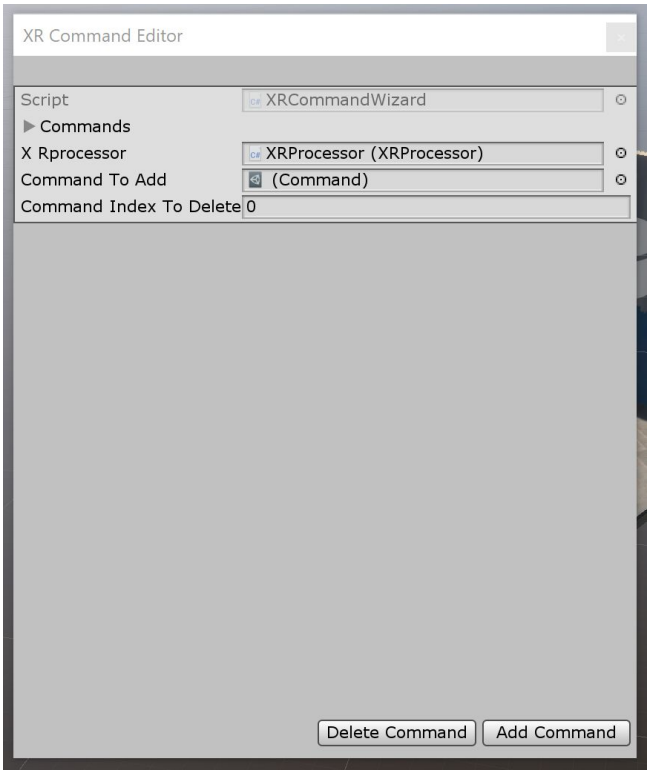
Unity 2019.2's XRInput system requires the user to keep lists of all the devices being tracked. These lists must be updated every frame because the devices might start or lose their connection to the computer after the program starts running. Alongside this, the number of devices per XRNode must be kept in check. A few problems arise with this: the developer will need to keep track of InputDevice lists in every script that requires input devices or keep them all in a central script that they will either need to shove all their input tests into or refer to with every script that needs XR input.

XRCS makes it so that all XRInput is processed by one MonoBehaviour component called the XR Processor. The XR Processor can have Commands attached to it (using the XR Command Editor) that will send XRInput values to VRCommand components that can process them.

In total, XRCS has 4 parts to it:

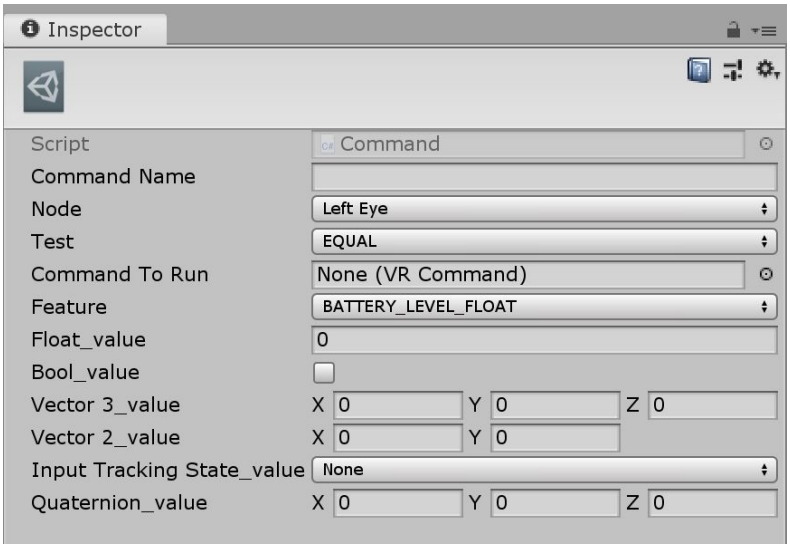
1. Setting up a scene for XRCS with an XR Processor
    a. Create an empty object in the scene and attach the XR Processor script ("XRProcessor.cs") component to it.
        i. This object will store all the Commands that will be run. It is recommended to not use more than one XR Processor component per scene.

2. Using the XR Command Editor
    a. Open XRInput/XR Command Editor from the top menu. This will open up the following wizard:



    b. If the XR Processor object is created correctly as described in Part 1 of this documentation, the component should automatically be loaded into the “X Rprocessor” field.
    c. To edit the new Command to be added, double click the empty Command object in the “Command To Add” field. Doing this should open up the following

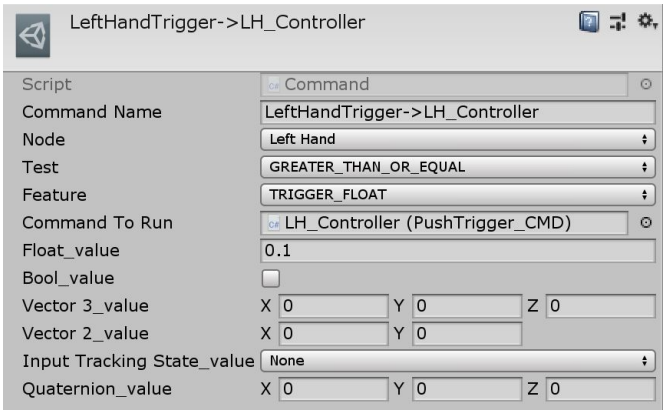
Command ScriptableObject in the Inspector:



d. Once the Command is ready to be added to the XR Processor, click the “Add Command” button at the bottom of the XR Command Editor.
e. To **delete any commands** from the current list of Commands in the XR Processor, open the “Commands” list by clicking on the triangle next to “Commands.”
    i. Find the element to be deleted and type its number into the “Command Index To Delete” box.
    ii. Press the “Delete Command” button on the bottom.
f. To **edit any existing commands** from the current list of Commands in the XR Processor, open the “Commands” list by clicking on the triangle next to “Commands.”
    i. Find the element to be edited and double click on it. This will open it up in the Inspector.
    ii. Make any edits necessary in the Inspector and the XRProcessor will update automatically.

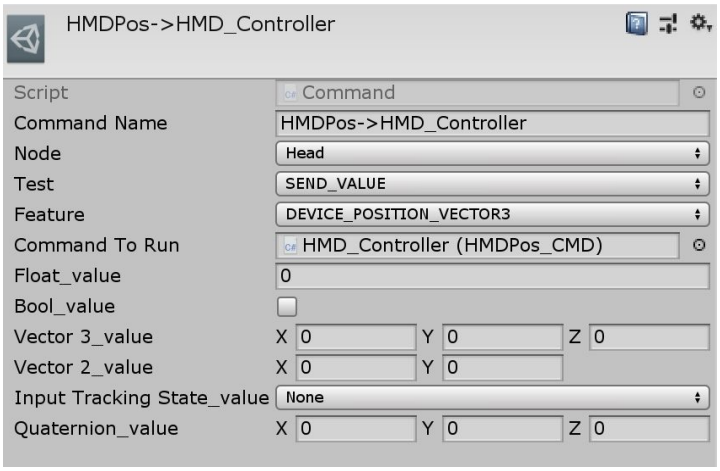
3. The Command object
    a. Command extends the ScriptableObject class. Commands are editable in the Inspector. They can also be created through the Asset/Create/ScriptableObjects/Command menu, however **this is not recommended**, and will be explained shortly.
    b. Command members:
        i. *Command Name*: This will be the name of the Command Scriptable Object. **Naming commands is highly recommended**, as the name will appear in the list of commands in the XR Command Editor. This will make it easier to tell which Command in the list does what.
        ii. *Node*: This refers to the XRNode the input is associated with (i.e. the **Input Device**)
        iii. *Test*: This is a test that is to be run on the input value before running the *Command To Run*. The result of *Test* is a value of type TestResult. The test result is sent to the *tr* parameter of *Command To Run*'s RunCommand() function.
        iv. *Feature*: This is the input to be tested on the device. Each feature in the enum has the data type the feature will output as a suffix.
        v. *Command To Run*: This accepts a VRCommand component. This component's RunCommand() function will be called every time the *Test* succeeds, or if *Test* is SEND_VALUE, RunCommand will be invoked every frame.
        vi. *Float_value*: The float value to run *Test* with if *Feature*'s suffix is _FLOAT.
        vii. *Bool_value*: The boolean value to run *Test* with if *Feature*'s suffix is _BOOL.
        viii. *Vector3_value*: The Vector3 value to run *Test* with if *Feature*'s suffix is _VECTOR3.
        ix. *Vector2_value*: The float value to run *Test* with if *Feature*'s suffix is _VECTOR2.
        x. *Input Tracking State_value*: The float value to run *Test* with if *Feature*'s suffix is _INPUTTRACKINGSTATE.
        xi. *Quaternion_value*: The Quaternion value to run *Test* with if *Feature*'s suffix is _QUATERNION.
    c. Sample Commands:

i. LeftHandTigger->LH_Controller



This Command invokes the RunCommand() function of the PushTrigger_CMD component of the LH_Controller GameObject every frame. If the Left Hand controller's Trigger float value is greater than or equal to 0.1, *tr* will be set to TestResult.TRUE. Otherwise, it will be set to TestResult.FALSE. It will also send the float value of the Trigger feature to the RunCommand() function's val_float.

ii. HMDPos->HMD_Controller



This command invokes the RunCommand() function of the HMDPos_CMD component of the HMD_Controller GameObject every frame. The *tr* parameter of the RunCommand() function will be set to TestResult.NA. It sends the Vector3 value of the Head's position to the RunCommand() function's val_vector3.

d. **Note**: Using the Asset Menu to create a Command Object and store it is not recommended because the *Command To Run* member needs to be a component of an object in the Scene. This is not easy to get working with the Asset Menu.

4. The VRCommand class and the RunCommand() function
    a. The VRCommand is an abstract class that extends MonoBehaviour. As such, it must be extended and then have its child be attached to an object.
        i. The class contains a single function, RunCommand() which has 10 parameters passed to it:
            1. *tr* is the test result if a test was performed.
                a. If *tr* is TestResult.NA, then no test was performed and the value was just sent.
                b. If *tr* is TestResult.TRUE, there was a test performed and it passed.
                c. If *tr* is TestResult.FALSE, there was a test performed and it failed.
            2. The rest of the *val_*-prefixed variables are the values sent from the Command being run. All but one of them will be set to default values when the function is invoked.
                a. 3.c.i's sample Command would be setting the *val_float* parameter to be the value of the TRIGGER_FLOAT feature of the Left Hand.
                b. 3.c.ii's sample Command would be setting the *val_vector3* parameter to be the value of the DEVICE_POSITION_VECTOR3 feature of the Head.
    b. To extend this class the RunCommand() function must be implemented.
        i. In this function use the *val_*-prefixed variable of the same type of the feature whose value will be sent by the expected Command.