



Immerseum

VR Simulator User Manual (version: BETA-0.8.1)

A Unity Editor Extension that provides a fast-and-easy way to simulate a virtual reality headset while in Unity's **Play Mode**. Accepts input from many types of input devices (including VR-specific controllers) and provides some basic movement support to help you test your VR scenes.

1 Copyright Page

This document and all material included herein is Copyright © Immerseum Inc. 2016. All rights are reserved.

Table of Contents

1.	Copyright Page	1
2.	Getting Started	3-6
2.1.	Getting Started with the VRSimulator	7-17
2.2.	Quick Start	18-21
2.3.	Requirements and Supported Devices	22-23
2.4.	FAQ	24-25
3.	Configuring the VRSimulator	26
3.1.	Managing Your Camera	27-29
3.2.	Managing Your Controllers	30-35
3.3.	Configuration Settings	36
3.3.1.	HMDSimulator Settings	36-37
3.3.2.	ControllerManager Settings	37-39
3.3.3.	MovementManager Settings	39-40
3.3.4.	InputActionManager Settings	40
4.	Input & Movement	41-45
4.1.	Custom Input Handling	46-51
4.2.	Custom InputAction Mapping	52-60
4.3.	Default Movement Mapping	61-66
4.4.	Default Input Actions	67-70
5.	Scripting Reference	71-72
5.1.	API Scripting Reference	73
5.2.	Event Reference	74-76
5.3.	Order of Events	77
6.	Release Notes	78-79
7.	Index	80

2 Welcome to the VR Simulator

Welcome to Immerseum's **VR Simulator**!

What does the VR Simulator do?

The **VR Simulator** is an editor extension for Unity to make developing virtual reality scenes easier when you don't have (or feel like hooking up) a VR headset. As the name suggests, it simulates a VR environment for you by:

- Adjusting your scene's camera position to simulate what it would look like if an HMD were controlling the camera,
- Moving and rotating your camera in response to keyboard, mouse, gamepad, etc. input to simulate walking and looking through an HMD,
- Simulating position-tracked controllers (even if they're not hooked up or currently being tracked) in your scene,
- Moving your simulated position-tracked controllers to simulate hand movement in your scene.

That's the the heart of it. However, the VR Simulator's real strength is its easy configurability and the ability to flexibly simulate your VR experience. Using the Unity Editor, you can adjust the VR experience you're simulating in a variety of ways:

- **Automatically detect headsets and controllers.** The VR Simulator automatically detects the headset you're using (and differentiates between SteamVR/Vive and Oculus Rift) and which inputs are currently connected to your development machine. Then, it only starts simulating if simulation is necessary: If you're connected to an HMD, you'll get the "real" experience and if not, you'll get the simulation.
- **Simulate different head heights.** Out of the box, the VR Simulator can simulate a **Seated**, **Standing**, or **Custom** head height to let you playtest seated, standing, and room-scale VR experiences.
- **Simulate controllers using primitives or prefabs.** Position-tracked controllers can either be simulated using configurable Unity primitives (spheres, cubes, etc.) or using any prefab. If your VR experience uses a custom controller model, you can use the same model in your simulations.
- **Configure simulated controllers' starting positions.** You can set one or both of your simulated controllers to any position you'd care to. Using default presets, you can put them at the player's (simulated) waist, at desk-level, reaching forward, or in a classic boxer's pose (put up your dukes!).
- **Move simulated controllers independently of each other.** Using simple scripting (seriously - it's one method call!) you can move each controller to one of the position presets or to any set of coordinates you want.
- **Test controller interaction with your scene.** You can configure your simulated controllers the same way you would configure your standard controller models. That way, you can attach [rigidbodies](#) and [colliders](#) to them and test their interactions with the rest of your VR scene.
- **Capture player input using events.** Easily test your existing interaction logic by subscribing to one of the VR Simulator's events - when the player pulls a trigger, have your scene do whatever it would normally do.

- **Easy C# scripting.** The **VR Simulator** provides an easy-to-work with C# scripting API that lets you build on, integrate with, or react to anything that happens in the VR Simulator.

What doesn't the VR Simulator do?

Be Careful!

The **VR Simulator** is meant to make development easier. It is **not** a production-grade asset, and is not designed to be included in the VR experiences that you release.

Can its input mapping and locomotion support be used in a production build? Sure, theoretically - but there are better ways of handling all of that and that is **not** what the **VR Simulator** was built to do.

The **VR Simulator** is not designed to provide "true" locomotion support for your VR scenes. It is meant to provide a quick-and-dirty approximation of locomotion support. As a result, it does not do any of the following:

- It has no specialized logic for graceful handling of inclines or stairs.
- It has no support for teleportation-based locomotion.
- It has no support for gaze-based input capture (selection or interaction).
- It hasn't been testing in zero-gravity scenes (though movement is gravity-adjusted in the configuration).
- It hasn't been tested in any mobile VR systems (e.g. Samsung Gear or Google Cardboard).

What HMDs and input devices does it support?

The **VR Simulator** is designed to work with:

- any SteamVR plugin-compatible HMD (i.e. the HTC Vive), and;
- the Oculus Rift.

Be Aware

Oculus Rift support has been tested with the **Oculus Rift CV1**. Presumably, it should work with the DK1 and DK2 insofar as the [Oculus Utilities for Unity](#) plugin works with those devices, but we haven't done any testing (because we don't have DK1s or DK2s).

We have not built support for OSVR's plugins into the VR Simulator - if you'd like to see OSVR support, [let us know!](#)

If you test the VR Simulator on any of these untested platforms, please [drop us a line](#) to let us know how it went!

The **VR Simulator** recognizes and supports input from the following devices:

- Keyboard
- Mouse
- Gamepad (XBox One Controller, specifically)
- HTC Wand (Vive Controllers)

- Oculus Remote
- Oculus Touch (theoretically! see note below)



Be Aware

Oculus Touch support has **not** been tested yet. That's because while the [Oculus Utilities for Unity](#) do provide an API for interacting with the Touch, we don't have any Touch controllers to test with.

If someone does have some Touch controllers and has tested the **VRSimulator** with them, please [drop us a line!](#)

What are its requirements?

The **VR Simulator**'s requirements are very simple:

- Windows or MacOS



You Should Know...

While the **VRSimulator** was specifically specifically built to support Windows and MacOS, it has only been tested under Windows.

- Unity 5.3 or higher,
- Both the [SteamVR plugin](#) or the [Oculus Utilities for Unity](#) package in your Unity project, and;
- One or both of the following camera rigs in your scene:
 - The **SteamVR**: [CameraRig] prefab, and/or;
 - The **Oculus**: OVRCameraRig prefab (or the OVRPlayerController prefab - which also contains the OVRCameraRig prefab)



You Should Know...

By default, you can find the camera rig prefabs in the following locations:

SteamVR: Assets / SteamVR / Prefabs /

Oculus: Assets / OVR / Prefabs /

And that's it!



Be Aware

If your Unity project or scene doesn't meet the requirements above, then the **VRSimulator** will throw an exception and log an error to the Unity console when you try to **Play** your VR scene.

How does it work?

Once you've imported the **VR Simulator** into your project, just drag the **[VRSimulator]** prefab into your VR scene - and that's it! You can then edit the **VR Simulator**'s configuration settings by clicking on the **[VRSimulator]** [gameObject](#) in your hierarchy.

You Should Know...

Once you've installed the VRSimulator, you can find sample scenes in your assets folder:

Assets / _ImmerseumSDK / 2_ExampleScenes /

For more details on working with the **VR Simulator**, please take a look at:

- **Getting Started with the VRSimulator (Section 2.1)**, and;
- The **QuickStart (Section 2.2)** guide.

2.1 Getting Started with the VR Simulator

Troubleshooting and Support

If you need help using the VR Simulator, we're here to help you! Reach out using any of the following:

- **Email** us at: support@immerseum.io
- Join our **Slack group**: immerseum-vr-dev.slack.com (you can get an invite by clicking here)
- **Tweet** at us: [@ImmerseumVR](https://twitter.com/ImmerseumVR)



Be Aware

We're a teensy, tiny team right now. The **VR Simulator** is a one-person project, so response times **will** vary. We'll do our best to get back to you ASAP, however.

Installing the VR Simulator

1. **Make sure your Unity project meets the requirements for the Immerseum VR Simulator (Section 2.3).**
2. **Download the Immerseum VR Simulator.**

Coming soon to the [Unity Asset Store](#)!



You Should Know...

While we're waiting for the **VR Simulator** to be made available in the Unity Asset Store, you're welcome to download a ZIP file with the asset files directly by clicking [this link](#).

3. **Extract the ZIP file to your hard drive.**



Best Practice

Remember where you extracted the ZIP file - you'll have to find the extracted folder to import it into Unity.

4. **Navigate to the folder where you extracted the ZIP file.** You should see two files in the folder: `vr-simulator-BETA-0.8.unitypackage` and `InputManager.asset`.
5. **If you will be creating a new project for the VR Simulator, then skip to step 7 for now. After you've completed the other steps, return to complete step 6.**
6. **Make sure that Unity is not currently running. Copy the InputManager.asset file to your VR project's Project Settings folder.** This will create new [Unity Input Manager](#) settings for you, defining the input manager axes and buttons that the VR Simulator will utilize.



Be Careful!

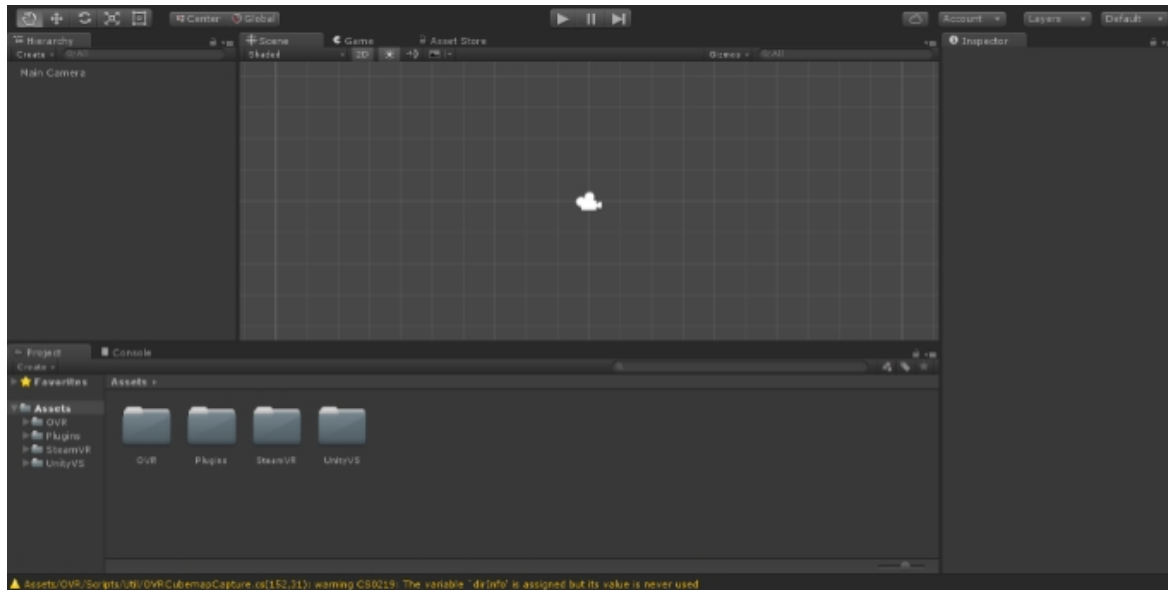
This step may over-write any existing input manager settings you may have created for your project.

If you really need to keep your own input manager settings, you'll need to turn off **Use Immerseum Defaults** in the VR Simulator's **InputActionManager** settings and define your own **%InputActions%** using **Custom InputAction Mapping**.

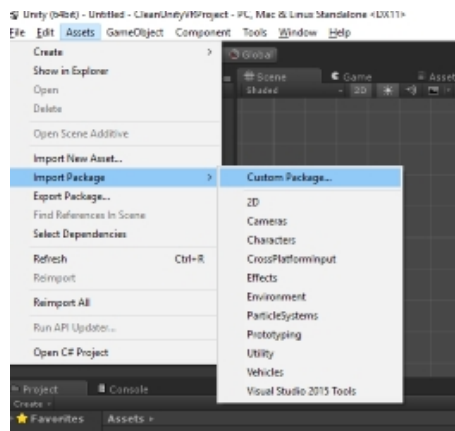
For More Information...

- InputAction Manager Settings (Section 3.3.4)
- Custom InputAction Mapping (Section 4.2)

7. In Unity, open your project (or create a new project).



8. Select Assets > Import Package > Custom Package...



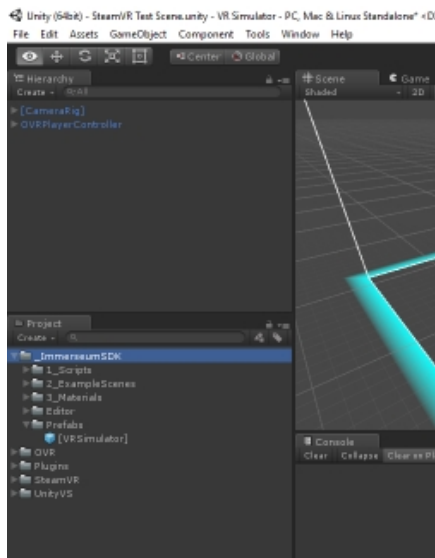
9. Find the folder where you extracted the ZIP file. Select the file named: VRSimulator-BETA-0.8.unitypackage and click Open.



Be Aware

If you're shown any prompts or warnings, just click to accept them. This will make sure that you apply the input mapping axis definitions in your [Unity Input Manager](#).

11. You should now see the _ImmerseumSDK folder in your Unity project's Assets folder. This is where the VRSimulator resides, where you can find its prefabs, scripts, and sample scenes.



12. **Did you copy over the** `InputManager.asset` **file into your VR project's** `ProjectSettings` **folder?** If not, then you can do so now by closing Unity and returning to **step #6** above.

Adding the VR Simulator to Your Scene

1. First, make sure your scene Hierarchy contains a -compatible camera rig:
 - the `SteamVR:[CameraRig]` prefab, and/or;
 - the `Oculus` camera rig, using either:
 - the `Oculus:OVRCameraRig` prefab, or;
 - the `Oculus:OVRPlayerController` prefab, or;

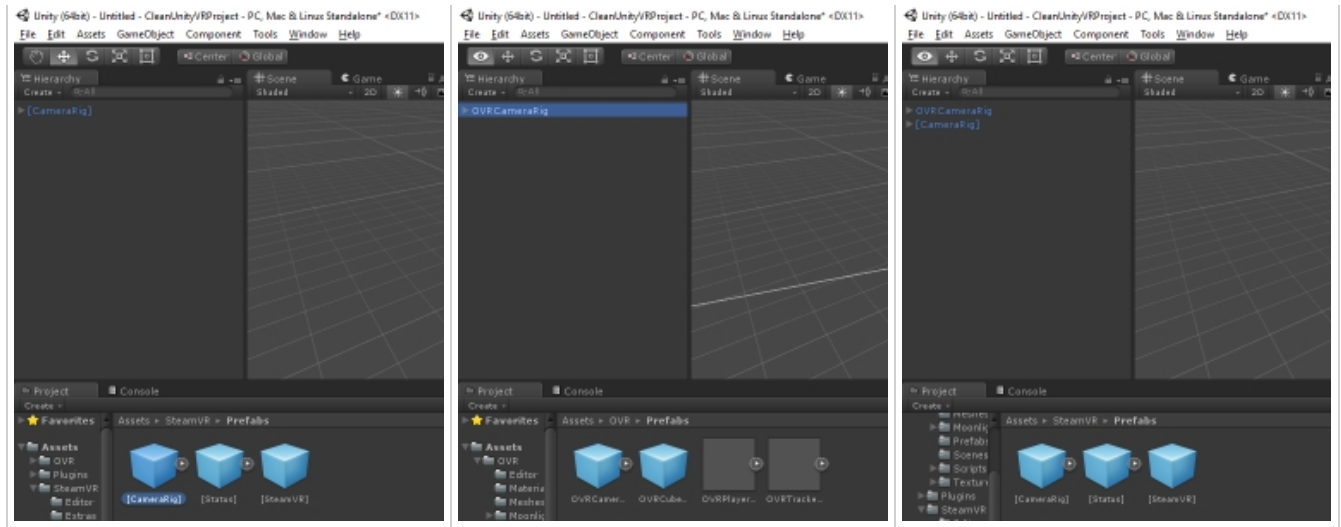
You Should Know...

By default, you can find the camera rig prefabs in the following locations:

SteamVR: `Assets / SteamVR / Prefabs /`

Oculus: `Assets / OVR / Prefabs /`

SteamVR:[CameraRig]	Oculus:OVRCameraRig	Both SteamVR / Oculus



You Should Know...

If your scene contains one active SteamVR camera rig and one active Oculus camera rig, then the VR Simulator will automatically use the SteamVR camera rig when simulating an HMD unless you configure the HMD Simulator's **Camera Rig** property.

2. From your project's Assets folder, find the **[VRSimulator]** prefab:

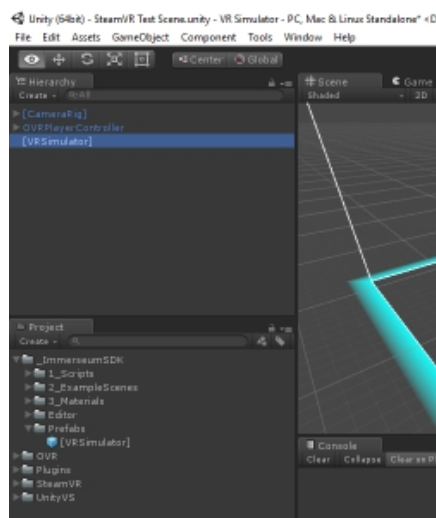
Assets/

_ImmerseumSDK/

Prefabs/

[VRSimulator]

3. Drag the **[VRSimulator]** prefab into your scene Hierarchy.



And that's it! The VR Simulator is now set up, and you can either use it right out of the box (just hit **Play**) or **Configure the VR Simulator (Section 3)** however you'd like.

What's in the Box?

The **Immerseum VRSimulator** contains one prefab: **[VRSimulator]**. You can find it in `Assets / _ImmerseumSDK / Prefabs /`

Attached to this prefab, you'll find four scripts:

- **HMDSimulator**. This is the VRSimulator's main script. It controls how the VRSimulator controls your camera, letting you configure what body position (Seated, Standing, Custom) you're simulating, whether to simulate controllers, and what type of object to use for controller simulation. [[Configuration](#) | [Scripting \('HMDSimulator Class' in the on-line documentation\)](#)]
- **ControllerManager**. This script lets you configure how your simulated controllers behave. It determines their starting position, configure their [colliders](#) and [rigidbodies](#), etc. [[Configuration](#) | [Scripting \('ControllerManager Class' in the on-line documentation\)](#)]
- **MovementManager**. This script lets you configure how your player/avatar moves in the scene. It also lets you disable Immerseum's default movement mapping if you want to supply your own. [[Configuration](#) | [Scripting \('MovementManager Class' in the on-line documentation\)](#)]
- **InputActionManager**. This script lets you disable Immerseum's default input mapping if you want to supply your own. [[Configuration](#) | [Scripting \('InputActionManager Class' in the on-line documentation\)](#)]



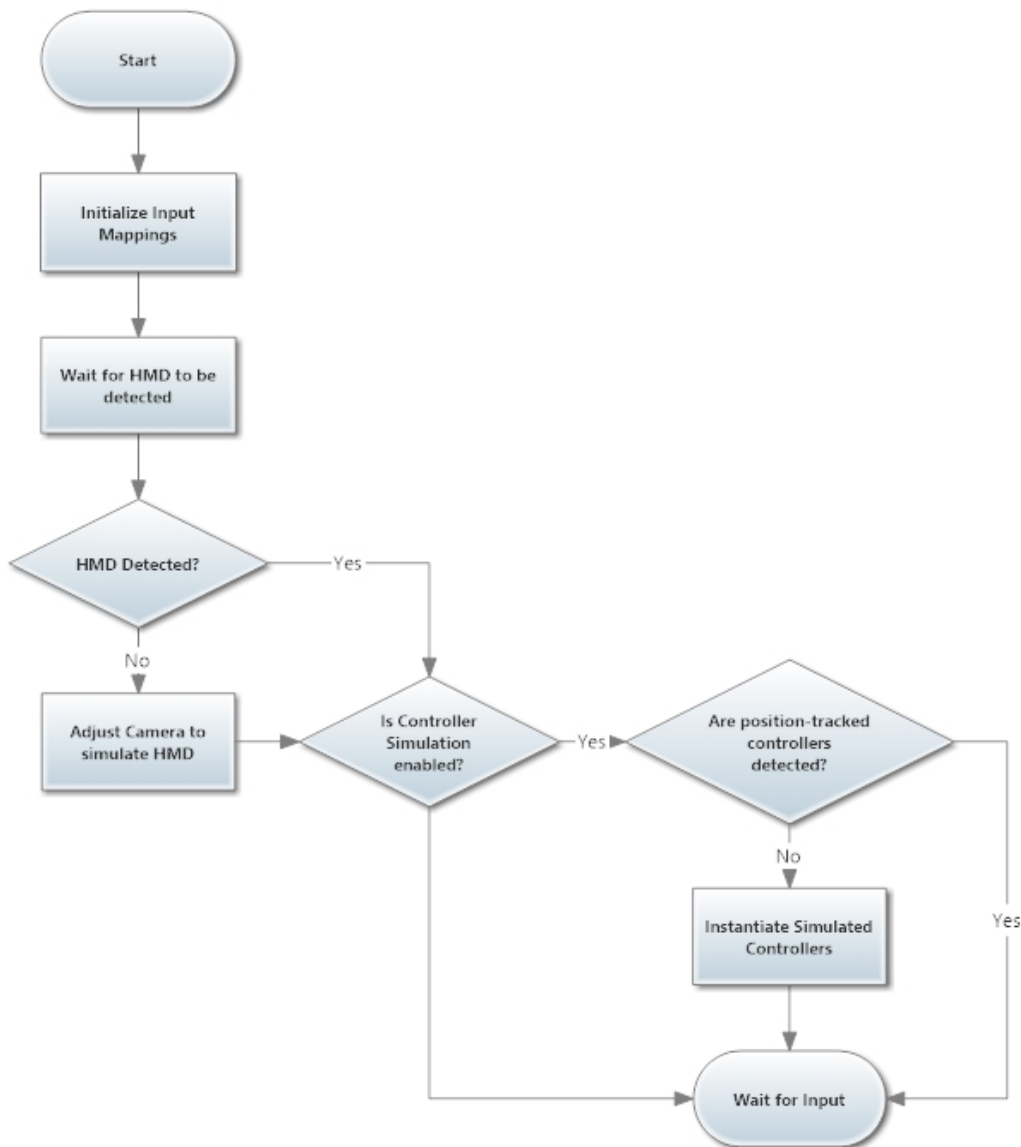
Best Practice

We've tried to be comprehensive about guidance in the editor. Every parameter also has a tooltip (just hover over it with your mouse), and you can also choose to **Display Help Boxes** (in the HMDSimulator) which will provide further guidance for you in each of the script editors.

The VRSimulator should run right out of the box - just enter **Play** mode in the Unity Editor.

How Does It Work?

Looked at from a high level, the VRSimulator generally follows the following path when your VR scene loads:



Then, whenever your player generates an Input Action, the VR Simulator fires the event for that Input Action. Any method that listens for the Input Action will then execute.

✓ Best Practice

Use Input Action Events to have your VR scene respond to your user's actions the way it would with a connected HMD / controllers. For example, you can create a method that listens for a trigger pull, and then moves the simulated controller when the user has pulled their trigger.

For More Information...

- Custom Input Handling (Section 4.1)

By default, the **MovementManager** listens for a standard set of Input Action Events, and if one of those events gets fired it will move the user/player avatar or rotate the camera based on the input.

You Should Know...

While it is an advanced technique, you can customize the way Input Actions are mapped to Movement. This means you can change which input buttons / thumbsticks / touchpads / etc. do what. This is an advanced technique, but with a little bit of scripting isn't that hard.

For More Information...

- [Custom InputAction Mapping \(Section 4.2\)](#)

For More Information...

- [Order of Events \(Section 5.3\)](#)
- [Configuring Input & Movement \(Section 4\)](#)
- [Custom Input Handling \(Section 4.1\)](#)
- [Custom Movement Mapping \(Section 4.3\)](#)

Configuring the HMDSimulator

Editor Settings

The **HMDSimulator** script exposes the following settings which are configurable within the Unity Editor:

Display Help Boxes

If enabled, will display helpful explanations in the Unity Editor for all VRSimulator scripts (disabled by default).

Log Input Actions

If enabled, will log any Input Action Event to the Unity Console (disabled by default).

Camera Rig

The Camera Rig that you wish to use for simulation. If left empty (default), will use whichever Camera Rig is present in your scene - if more than one is present, will default to **SteamVR**: [CameraRig].

HMD Discovery Time

Neither the [SteamVR plugin](#) nor [Oculus Utilities for Unity](#) discover a connected HMD right away. It always takes a little time - maybe a few milliseconds, maybe a second or two - depending on your motherboard, graphics card, the resources on your system, etc. This setting lets you configure how many seconds to wait for an HMD before giving up and simulating the HMD. (default: 2 seconds)



Be Aware

On our testing rigs, we've found that an HMD will typically be discovered in about 1.2 seconds if it's connected. But as this has varied widely (even on the same machine), we've found that 2 seconds (our default value) works well as a good compromise.

Your mileage may vary!

Height

This is the height / position that you wish to simulate. Your options are:

- **Standing (default)** - uses User Profile height if available, otherwise defaults to 1.755 (which for Oculus sets eye-height at approximately 1.65).
- **Seated** - uses User Profile data if available, otherwise defaults 1.06.
- **Custom** - If selected, you can enter your own targeted height at whatever value you choose.

Simulate Controllers

If enabled, will simulate controllers if no HMD is detected. (default: disabled)

Controller Primitive

This is the Unity [primitive](#) type that you wish to use when simulating controllers. Your options are:

- **Cube**
- **Sphere** (default)
- **Cylinder**
- **Capsule**

- **Custom** - If selected, you can select any prefab from your Assets folder. You can use different prefabs for left-hand right controllers if you want.

Controller Scaling

Determines the size of your simulated controller. Assumes each primitive has a size of (1, 1, 1) and scales them up or down appropriate. If using a custom controller primitive, scaling is determined by the prefab itself. (default: 0.05)

For More Information...

- [Managing Your Camera \(Section 3.1\)](#)
- [Managing Your Controllers \(Section 3.2\)](#)

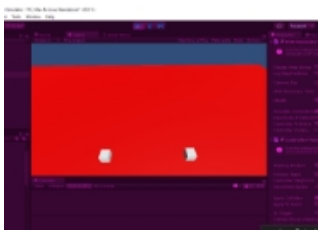


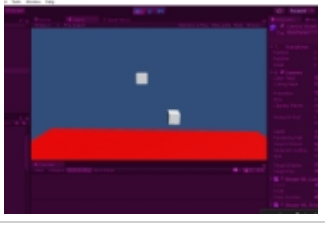
Configuring the ControllerManager

Editor Settings

The **ControllerManager** script exposes the following settings which are configurable within the Unity Editor:

Starting Position

This is the positon / posture in which your simulated controllers will start. Your options are:

Origin	Simulates "dangling at your waist". They start at more-or-less waist height (lower if seated) and slightly forward of your player avatar's implied body. This slight forward simulation is because players rarely have controllers at rest when in a VR experience, and this lets you more accurately see where your controllers are when in simulation mode.	
Forward (default)	Simulates "resting on your desk/keyboard". They are at about torso-height and more substantially forward.	
Reaching	Simulates hands extended out from the shoulder.	
Boxer	Simulates a "puglist's pose" with the hands near eye-level, with one (the primary) hand higher and more forward than the secondary hand.	
Prefab	Uses whatever position automatically derives from the simulated controller's primitive (useful if you're using a custom prefab to simulate a controller).	



Be Aware

When using **Play** mode in Unity, your simulated controllers might seem really small or really far away. That's an artifact of how the SteamVR plugin and Oculus Utilities for Unity compositors render your VR scene to a regular (non-HMD) monitor. Basically, you're getting some "lens warping" effects when going from a stereoscopic view (in your HMD) to a non-stereoscopic view (on your monitor).

By design, the VR Simulator does not adjust for that so as to maintain positional, rotational, and scaling consistency with your actual VR scene. In other words, so long as you're positioning things relative to your simulated controllers, they should be positioned "properly" relative to your user's controllers when you're actually wearing an HMD.

Primary Hand

This determines which hand is positioned the furthest-forward when in **Boxer** position. It is also exposed programmatically, letting you give your player "handedness". Your options here are:

- **Left** (default)
- **Right**
- **Ambidextrous** - both hands are treated the same

Controller Height Adjustment

This determines the distance by which the controllers should be offset relative to head height. This is a "magic number" used in downstream calculations - adjust with care! (default: 0.48)

Movement Speed

This determines the rate at which simulated controllers move when moving independently of the user's body. (default: 0.5)

Apply Colliders

If enabled, applies a [Collider](#) to the controller selected by **Apply to Hand**. (default: disabled)



You Should Know...

If **Apply Colliders** is enabled, you can set the same [Collider](#) properties as you would if you were attaching a [Collider](#) component to a [GameObject](#). These properties will appear right below the "**Apply Colliders**" setting, and then be applied to whichever hand is indicated by the **Apply to Hand** setting.



Be Careful!

If you are using a Custom controller primitive, **Apply Colliders** and related settings will **override** any [colliders](#) that have been attached to your custom controller prefab.

Apply to Hand

Indicates which simulated controllers should get the [colliders](#). Your options are:

- **Left** - Only the left-hand controller will get a collider.
- **Right** - Only the right-hand controller will get a collider.
- **Ambidextrous (default)** - Both controllers will get a collider (with identical settings).

Apply Rigidbody

If enabled, applies a [Rigidbody](#) to the controller selected by **Apply to Hand**. (default: disabled)



You Should Know...

If **Apply Rigidbody** is enabled, you can set the same [Rigidbody](#) properties as you would if you were attaching a [Rigidbody](#) component to a [GameObject](#). These properties will appear right below the "**Apply Rigidbody**" setting, and then be applied to whichever hand is indicated by the **Apply to Hand** setting.

Be Careful!

If you are using a Custom controller primitive, **Apply Rigidbody** and related settings will **override** any [rigidbodies](#) that have been attached to your custom controller prefab.

Apply to Hand

Indicates which simulated controllers should get the [colliders](#). Your options are:

- **Left** - Only the left-hand controller will get a collider.
- **Right** - Only the right-hand controller will get a collider.
- **Ambidextrous (default)** - Both controllers will get a collider (with identical settings).

[For More Information...](#)

- [Managing Your Controllers \(Section 3.2\)](#)

Configuring the MovementManager

Editor Settings

The **MovementManager** script exposes the following settings which are configurable within the Unity Editor:

Default Input Map

If enabled, uses Immerseum's **default movement mapping (Section 4.3)**. (default: enabled)

Be Careful!

If you disable the **Default Input Map** you will have to provide your own movement mapping through scripting.

[For More Information...](#)

- [Custom Input Handling \(Section 4.1\)](#)

Is Strafe Enabled

If enabled, movement using an input device's x-axis moves the player/user laterally (side-to-side / strafe). If disabled, movement using an input device's x-axis rotate's the camera from side-to-side (yaw). (default: enabled)

Is Run Enabled

If enabled, the player can run (move at twice the default speed). (default: true)

Is Run Active

If enabled, the player is currently running (movement speed is effectively doubled). (default: false)

Gamepad Primary Trigger

This determines which Gamepad Trigger is considered the "primary" one (by default, primary and secondary triggers are mapped differently and so can do different things). Your options are:

- **Left**
- **Right** (default)

Advanced Settings

Be Careful!

The following are advanced settings that have a significant impact on user experience. Adjust them with caution.

Rotation Ratchet

Sets the number of degrees to rotate the user when a ratchet-rotation input action occurs. (default: 45)

Acceleration Rate

Sets the rate at which the user accelerates when starting movement. (default: 0.1)

Forward Damping Rate

Sets the rate by which forward movement is dampened/smoothed. (default: 0.3)

Back/Side Damping Rate

Sets the rate by which backwards and lateral is dampened/smoothed. (default: 0.3)

Gravity Adjustment

Sets the adjustment that occurs to gravity while the player is in motion. (default: 0.379)

For More Information...

- [Configuring Input & Movement \(Section 4\)](#)
- [Immerseum Movement Mapping \(Section 4.3\)](#)
- [Custom Input Handling \(Section 4.1\)](#)

Configuring the InputActionManager

Editor Settings

The **InputActionManager** script exposes the following settings which are configurable within the Unity Editor:

Use Immerseum Defaults

If enabled, applies Immerseum's **default Input Actions (Section 4.4)**. This determines which combination of input buttons, axes, etc. on your different input devices do the same things.

Be Careful!

If you disable Immerseum's default Input Actions but still want to use the VR Simulator's movement management system, you will need to supply your own custom Input Actions. This will require scripting.

For More Information...

- [Default Input Actions \(Section 4.4\)](#)
- [Custom InputAction Mapping \(Section 4.2\)](#)
- [Configuring Input & Movement \(Section 4\)](#)

For More Information...

- [Configuring Input & Movement \(Section 4\)](#)
- [Default Input Actions \(Section 4.4\)](#)
- [Custom InputAction Mapping \(Section 4.2\)](#)
- [Default Movement Mapping \(Section 4.3\)](#)
- [Custom Input Handling \(Section 4.1\)](#)

Scripting and the VR Simulator

The **VR Simulator** is designed to be integrated into your scripts quickly and easily, so as to help you truly simulate player interaction in your VR scene(s). As a result, it has an extensive and flexible API built into it that lets you do lots of things:

- Move the simulated controllers whenever (and wherever) you want.
- Detect input from the user across all of the supported input devices.
- Customize how input buttons / axes map to Input Actions.
- Customize how input actions map to movement.

Please be sure to take a look at our extensive **Scripting Reference (Section 5)** for more information.

2.2 Quick Start

Install the Immerseum VRSimulator

1. **Make sure your Unity project meets the requirements for the Immerseum VRSimulator (Section 2.3).**
2. **Download the Immerseum VRSimulator.**

Coming soon to the [Unity Asset Store](#)!

You Should Know...

While we're waiting for the **VRSimulator** to be made available in the Unity Asset Store, you're welcome to download a ZIP file with the asset files directly by clicking [this link](#).

3. **Extract the ZIP file to your hard drive.**

Best Practice

Remember where you extracted the ZIP file - you'll have to find the extracted folder to import it into Unity.

4. **Navigate to the folder where you extracted the ZIP file.** You should see two files in the folder: `vr-simulator-BETA-0.8.unitypackage` and `InputManager.asset`.
5. **If you will be creating a new project for the VRSimulator, then skip to step 7 for now. After you've completed the other steps, return to complete step 6.**
6. **Make sure that Unity is not currently running. Copy the InputManager.asset file to your VR project's Project Settings folder.** This will create new [Unity Input Manager](#) settings for you, defining the input manager axes and buttons that the VRSimulator will utilize.

Be Careful!

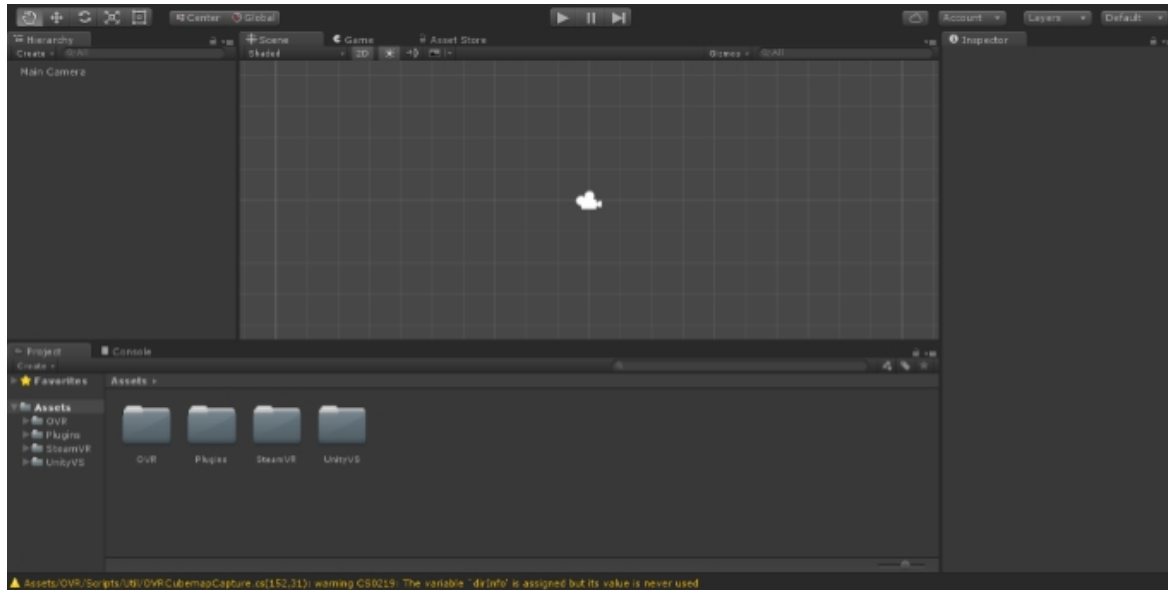
This step may over-write any existing input manager settings you may have created for your project.

If you really need to keep your own input manager settings, you'll need to turn off **Use Immerseum Defaults** in the VRSimulator's **InputActionManager** settings and define your own **%InputActions%** using **Custom InputAction Mapping**.

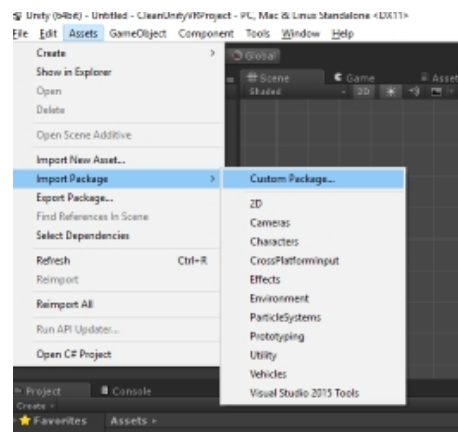
For More Information...

- [InputAction Manager Settings \(Section 3.3.4\)](#)
- [Custom InputAction Mapping \(Section 4.2\)](#)

7. **In Unity, open your project (or create a new project).**



8. Select Assets > Import Package > Custom Package...



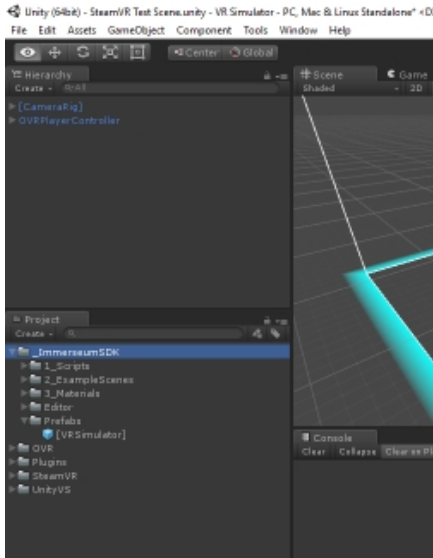
9. Find the folder where you extracted the ZIP file. Select the file named: VRsimulator-BETA-0.8.unitypackage and click Open.



Be Aware

If you're shown any prompts or warnings, just click to accept them. This will make sure that you apply the input mapping axis definitions in your [Unity Input Manager](#).

11. You should now see the `_ImmerseumSDK` folder in your Unity project's Assets folder. This is where the VRsimulator resides, where you can find its prefabs, scripts, and sample scenes.



12. **Did you copy over the** `InputManager.asset` **file into your VR project's** `ProjectSettings` **folder?** If not, then you can do so now by closing Unity and returning to **step #6** above.

Open a Sample Scene

The **VRSimulator** comes packaged with a sample scene which you can find in:

```
Assets /
    _ImmerseumSDK /
        2_ExampleScenes /
```

Add the VRSimulator to Your VR Scene

1. First, make sure your scene Hierarchy contains a -compatible camera rig:
 - the `SteamVR:[CameraRig]` prefab, and/or;
 - the Oculus camera rig, using either:
 - the `Oculus:OVRCameraRig` prefab, or;
 - the `Oculus:OVRPlayerController` prefab, or;

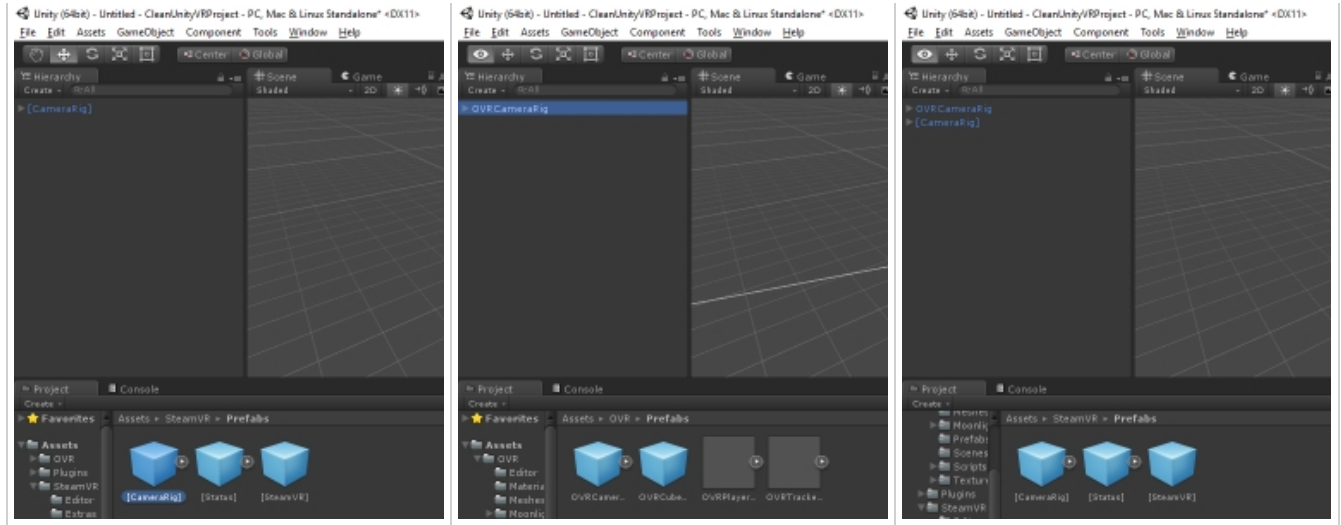
You Should Know...

By default, you can find the camera rig prefabs in the following locations:

SteamVR: `Assets / SteamVR / Prefabs /`

Oculus: `Assets / OVR / Prefabs /`

SteamVR:[CameraRig]	Oculus:OVRCameraRig	Both SteamVR / Oculus
---------------------	---------------------	-----------------------



You Should Know...

If your scene contains one active SteamVR camera rig and one active Oculus camera rig, then the VR Simulator will automatically use the SteamVR camera rig when simulating an HMD unless you configure the HMD Simulator's **Camera Rig** property.

2. From your project's Assets folder, find the **[VRSimulator]** prefab:

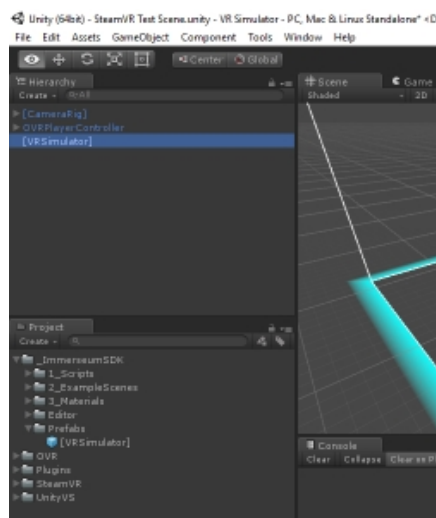
Assets/

_ImmerseumSDK/

Prefabs/

[VRSimulator]

3. Drag the **[VRSimulator]** prefab into your scene Hierarchy.



And that's it! The VR Simulator is now set up, and you can either use it right out of the box (just hit **Play**) or **Configure the VR Simulator (Section 3)** however you'd like.

2.3 Requirements and Supported Devices

Requirements

The **VR Simulator**'s requirements are very simple:

- Windows or MacOS

You Should Know...

While the **VR Simulator** was specifically specifically built to support Windows and MacOS, it has only been tested under Windows.

- Unity 5.3 or higher,
- Both the [SteamVR plugin](#) or the [Oculus Utilities for Unity](#) package in your Unity project, and;
- One or both of the following camera rigs in your scene:
 - The **SteamVR**: [CameraRig] prefab, and/or;
 - The **Oculus**: OVRCameraRig prefab (or the OVRPlayerController prefab - which also contains the OVRCameraRig prefab)

You Should Know...

By default, you can find the camera rig prefabs in the following locations:

SteamVR: Assets / SteamVR / Prefabs /

Oculus: Assets / OVR / Prefabs /

And that's it!

Supported HMDs

The **VR Simulator** is designed to work with:

- any SteamVR plugin-compatible HMD (i.e. the HTC Vive), and;
- the Oculus Rift.

Be Aware

Oculus Rift support has been tested with the **Oculus Rift CV1**. Presumably, it should work with the DK1 and DK2 insofar as the [Oculus Utilities for Unity](#) plugin works with those devices, but we haven't done any testing (because we don't have DK1s or DK2s).

We have not built support for OSVR's plugins into the VR Simulator - if you'd like to see OSVR support, [let us know!](#)

If you test the VR Simulator on any of these untested platforms, please [drop us a line](#) to let us know how it went!

Supported Input Devices

The **VR Simulator** recognizes and supports input from the following devices:

- Keyboard
- Mouse
- Gamepad (XBox One Controller, specifically)
- HTC Wand (Vive Controllers)
- Oculus Remote
- Oculus Touch (theoretically! see note below)



Be Aware

Oculus Touch support has **not** been tested yet. That's because while the [Oculus Utilities for Unity](#) do provide an API for interacting with the Touch, we don't have any Touch controllers to test with.

If someone does have some Touch controllers and has tested the **VRSimulator** with them, please [drop us a line](#)!

2.4 FAQ

What VR platforms do you support?

The **VR Simulator** is designed to work with:

- any SteamVR plugin-compatible HMD (i.e. the HTC Vive), and;
- the Oculus Rift.



Be Aware

Oculus Rift support has been tested with the **Oculus Rift CV1**. Presumably, it should work with the DK1 and DK2 insofar as the [Oculus Utilities for Unity](#) plugin works with those devices, but we haven't done any testing (because we don't have DK1s or DK2s).

We have not built support for OSVR's plugins into the VR Simulator - if you'd like to see OSVR support, [let us know!](#)

If you test the VR Simulator on any of these untested platforms, please [drop us a line](#) to let us know how it went!

So you don't support Google Daydream, PlayStation VR, HoloLens, Meta, etc?

Not yet. However, Google Daydream and PlayStation VR are both on our roadmap.

As for augmented reality? The VR Simulator isn't likely to support any of those platforms in the foreseeable future.

What about mobile VR?

Our development focus has been on fully-immersive HMD VR. Therefore, we haven't done any testing on any mobile VR platforms (Google Cardboard, Samsung Gear, etc.).

Insofar as the Oculus Utilities for Unity are compatible with Samsung Gear, it is possible that the VR Simulator will work. But we haven't tested it. If you try it, [drop us a line](#) to let us know how it went!

Are you planning to make the VR Simulator available for Unreal Engine, Cryengine, Lumberyard, <insert dev platform here>?

Nope. Nothing against any of those great tools, but we believe strongly that Unity offers the best solution for long-term VR development from both technical and business perspectives.

Barring some huge shifts in the marketplace, the VR Simulator (and the rest of the Immerseum SDK) will be available on Unity.

Are you planning to expand locomotion support to include teleportation, walk-in-place, etc.?

Well...not really. We've got some interesting multi-modal solutions for locomotion that we're building into the Immerseum SDK, but de-coupling them from the broader SDK would be a complex undertaking. By contrast, the VRSimulator in its current form is something that we were able to pretty quickly carve out and make independent.

Let us know if you'd like to see multi-modal, cross-platform, and configurable locomotion - if there's a lot of demand for it, we might come to the conclusion that it's worth the effort to carve it out and make it available.

How about raycasting and gaze interactions?

It's not really a priority for us. Theoretically, we can do it - carving that functionality out of our broader SDK and making it work independently the way the VRSimulator does. But it's non-trivial work to do, and we're not sure we'd be justified taking the time to do it.

Let us know if you'd like to see configurable and multimodal raycasting and gaze-based input models in the VRSimulator. If a lot of people are looking for that, we might decide it's worth taking the time to carve it out and release it.

In the meantime, it is quite simple to integrate your own raycasting / gaze-based interaction system with the VRSimulator. Essentially, you just need to listen to the InputAction Events and have your code respond accordingly.

For More Information...

- **Custom Input Handling (Section 4.1)**

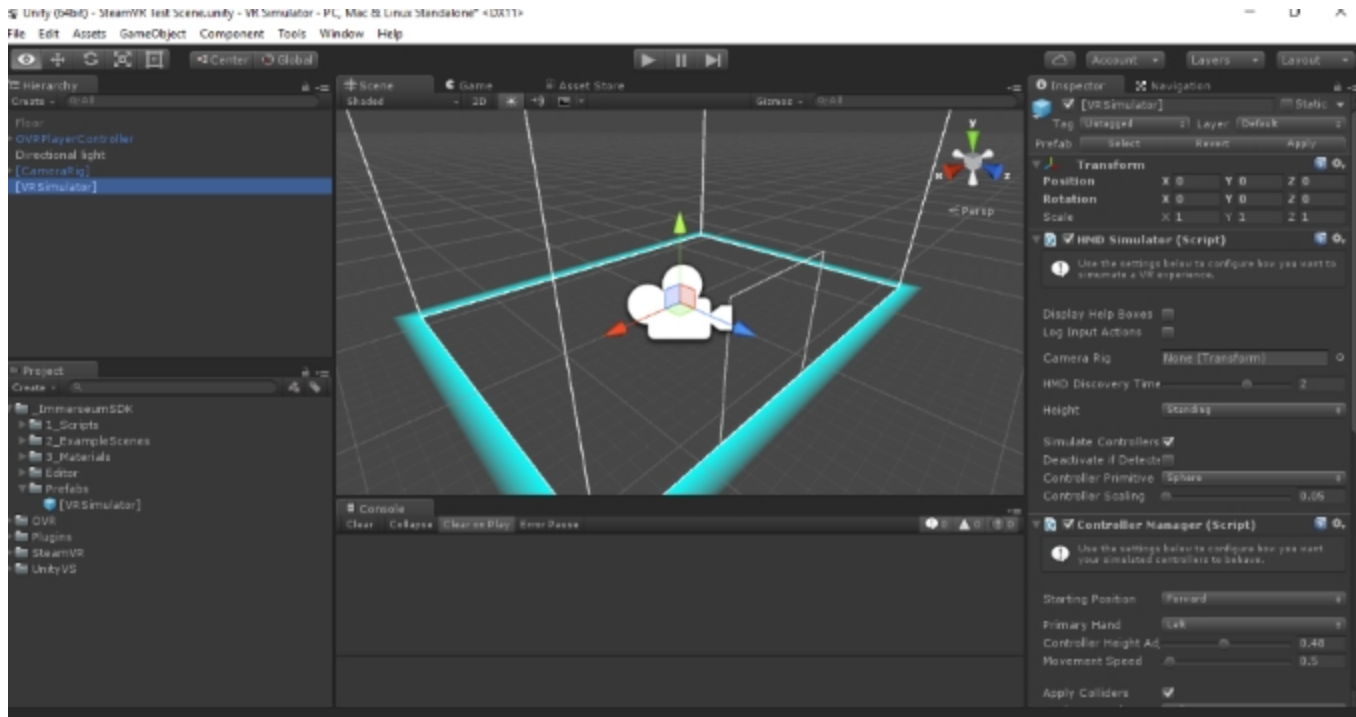
Are you planning to include UI interaction or free-form input support (e.g. typing, voice-activated movement, etc)?

So the VRSimulator doesn't really have anything to do with UI. From the VRSimulator's perspective, UI gameobjects are any other kind of gameobject. That means that you can simulate controller or camera interaction with UI elements just as you can simulate their interaction with an in-world object in your VR scene. Nothing special needed.

However, UI can often feature more complicated modes of interaction (typing, etc.). We're not planning to extend support in that direction. However, if you want to use the VRSimulator's input/movement system to capture and react to additional inputs, it's quite easy to extend the system using **Custom Input Handling (Section 4.1)**.

3 Configuring the VR Simulator

The **VR Simulator** is controlled almost entirely through the [Unity Inspector](#). To adjust any of the VR Simulator's configuration settings, simply select the **[VR Simulator]** gameobject in your Scene hierarchy. On the Unity Inspector pane you'll see the four scripts that control the VR Simulator's behavior:



These scripts allow you to:

- **Manage your camera in the VR scene (Section 3.1), and;**
- **Manage how controllers are simulated in the VR scene (Section 3.2), and;**
- **Configure how the player avatar moves in the VR scene (Section 4),**
- **Turn default input and movement mapping on and off (Section 4).**

Be Careful!

If you disable default input mapping or default movement mapping, you will need to supply your own using custom scripting. This is an advanced technique.

For More Information...

- **Default Input Mapping (Section 4.4)**
- **Default Movement Mapping (Section 4.3)**
- **Custom Input Handling (Section 4.1)**
- **Custom Movement Mapping (Section 4.2)**

3.1 Managing Your Camera

One of the challenges in developing for VR is the fact that you have limited control over your user's environment. Maybe you're expecting them to be sitting down, but in reality they'll be standing. Maybe you'll expect them to be standing, and in reality they'll be seated on the floor. Maybe their room-scale play space is 20x20, or maybe it's 5x5. That adds a level of complexity to the development and testing experience that is...well, a pain.

The **VRSimulator** helps make testing for different situations a little easier. Key to that process is effectively managing your VR scene's camera. The VRSimulator uses your VR Camera Rig as a proxy for the user's head/eyes. That's exactly what the VR HMD does when it's connected to your computer: It tracks where the HMD is, and adjusts the position and rotation of the Camera Rig based on its tracked position.

Using the VRSimulator and the **HMD Simulator Configuration Settings (Section 3.3.1)**, you can:

- Simulate your user's position at any height.
- Examine how your scene plays on the SteamVR or Oculus Rift platform.



Be Aware

Always remember that the **VRSimulator** is using your virtual reality camera rigs to display VR content on your regular (non-HMD) monitor.

By design, both the [SteamVR plugin](#) and [Oculus Utilities for Unity](#) optimize rendering for a stereoscopic display in an HMD. The compositors they use to render monoscopically in Unity Play Mode create varying degrees of visual distortion, creating a sort of fish-eye lens effect.

This will make objects in your VR scene look strange - sometimes very close, sometimes far away - when you view them on your screen. If you put your HMD on and take it off to look at your on-screen view, the distortions are pretty apparent. This is particularly apparent when you look at the simulated controllers generated by the VRSimulator.

By design, the **VRSimulator** does nothing to compensate for this distortion - it just uses the VR camera rigs as they are, and positions its simulated controllers in worldspace so that they would "look right" rendered in your VR HMD.

This is an intentional design choice, meant to maintain positional, rotational, and scaling consistency between your simulated VR environment and your actual VR environment. If we were to position the simulated controllers so that they seemed undistorted on a regular monitor, then they would not interact with the rest of your VR scene the way your player's actual VR controllers would.

Sorry about the distortion - but the consistency in design and positioning for your actual VR experience is really worth it.

Setting Up Your VR Camera Rig(s)

Before you can use the **VRSimulator**, you need to set up your camera rig. The **VRSimulator** supports two camera rig prefabs:

- **SteamVR:[CameraRig]**, which can be found in: `Assets / SteamVR / Prefabs /`
- **Oculus:OVRCameraRig**, which can be found in: `Assets / OVR / Prefabs /`

You Should Know...

The **Oculus:OVRCameraRig** prefab can either be inserted directly from Assets / OVR / Prefabs /, or you can also insert the **Oculus:OVRPlayerController** prefab.

You can set the **HMD Simulator's Camera Rig** setting equal to either **Oculus:OVRCameraRig** directly, or to its **OVRPlayerController** parent, depending on your preferences.

Your VR scene needs to contain one or more of these prefabs. It also cannot contain the default Unity **Main Camera** gameobject.

If your scene contains only one of these camera rig prefabs, the VR Simulator will automatically use it when simulating an HMD. However, if your scene contains more than one camera rig prefab, then:

- **You can set the VR Simulator to simulate using a specific camera rig.** Drag the camera rig [gameobject](#) from your scene hierarchy to the **Camera Rig** setting in the **HMD Simulator (Section 3.3.1)** script (attached to the **[VR Simulator]** gameobject). Now, when you hit **Play** in Unity, the VR Simulator will simulate that HMD when there's no HMD connected, and disable any others your scene might contain.
- **You can use VR Simulator's default camera rig.** If there are multiple camera rig prefabs in your scene, and one of them is the **SteamVR: [CameraRig]**, and you have left the **HMD Simulator (Section 3.3.1)**'s **Camera Rig** setting blank, then the VR Simulator will by default use the **SteamVR: [CameraRig]** when simulating your HMD and disable any other camera rig prefabs in your scene.

Be Careful!

If your scene does **not** contain **SteamVR: [CameraRig]**, but does contain multiple instances of Oculus camera rigs (for example, one **Oculus:OVRPlayerController** and a sibling **Oculus:OVRCameraRig** or multiple **OVRCameraRig** objects), then you're going to get strange behavior (both in the VR Simulator, and in your Oculus HMD itself).

You should only ever have one instance of **OVRCameraRig** enabled in any VR scene.

Discovering an HMD

If the **VR Simulator** detects a connected HMD, then it will not simulate an HMD (because it assumes you'll be using your headset).

However, detecting a connected headset does not happen instantly. Both the SteamVR plugin and the Oculus Utilities for Unity take a little time to discover a connected headset. The amount of time varies widely, we suspect based on motherboard, graphics card, system resources, etc. That's why the VR Simulator automatically waits a set time to discover a headset before activating your simulations. You can configure the amount of time to wait in the **HMD Simulator (Section 3.3.1)**'s **HMD Discovery Time** setting.

You Should Know...

So long as the VR Simulator is in your scene, it will still capture and apply your Input & Movement controls. So you can still explore your scene with your headset connected but not on your head - the inputs will all work just the same. You might just have a bit of a funny view, depending on where your headset is.

Setting Your Player's Position / Camera's Height

When actually using a headset, your camera's height is automatically calculated by your VR platform's sensors and your user profile. Obviously, that won't work if there's no HMD connected. Therefore, the VR Simulator lets you set the height that you wish to simulate using the **HMD Simulator (Section 3.3.1)**'s **Height** setting.

You can set your user's initial height to one of three options:

- **Standing** - This simulates standing or room-scale VR. It takes the statistical average human height (1.775) and adjusts for standard eye level differences (setting eye level to approximately 1.65).
- **Seated** - This simulates a seated posture, significantly lower than the standing height mentioned above (set to approx. 1.06).
- **Custom** - This lets you set your own custom height to simulate whatever height you wish.



Be Aware

The **Height** setting only sets your camera's **initial** height. What you do with your simulated camera's height later on is entirely up to you.

3.2 Managing Your Controllers

One of the great things about VR is its ability to bring your hands into an interactive, immersive, digital world. That creates lots of great, immersive new design opportunities for us. But if we don't have our position-tracked controllers with us, or if we're working somewhere we can't wildly swing our arms around, that can make developing and testing things a little difficult. And then when we realize that the SteamVR plugin doesn't render controller models if there are no controllers detected, and that Oculus Utilities for Unity plugin doesn't render controller models at all - well, all of that can be a bit of a pain.

The VR Simulator helps with that by generating visible controller models and positioning them more-or-less where you might expect a player's hands to be. You can then use these simulated controllers to emulate player movement and test your interactions. For example, do you want your controllers to knock things off of surfaces in a particular way? Well, just simulate a controller and knock it into something.

If enabled, the **VR Simulator**:

- Generates new gameobjects using primitives or prefabs for your controllers.
- Applies [colliders](#) and [rigidbodies](#) to your simulated controllers to make them interact with your world the way your real controllers would.
- Lets you position the controllers in natural ways using simple presets (no need to mess with coordinates!).
- Lets you programmatically move controllers to a set of coordinates if and when you need to.



Be Aware

Always remember that the **VR Simulator** is using your virtual reality camera rigs to display VR content on your regular (non-HMD) monitor.

By design, both the [SteamVR plugin](#) and [Oculus Utilities for Unity](#) optimize rendering for a stereoscopic display in an HMD. The compositors they use to render monoscopically in Unity Play Mode create varying degrees of visual distortion, creating a sort of fish-eye lens effect.

This will make objects in your VR scene look strange - sometimes very close, sometimes far away - when you view them on your screen. If you put your HMD on and take it off to look at your on-screen view, the distortions are pretty apparent. This is particularly apparent when you look at the simulated controllers generated by the VR Simulator.

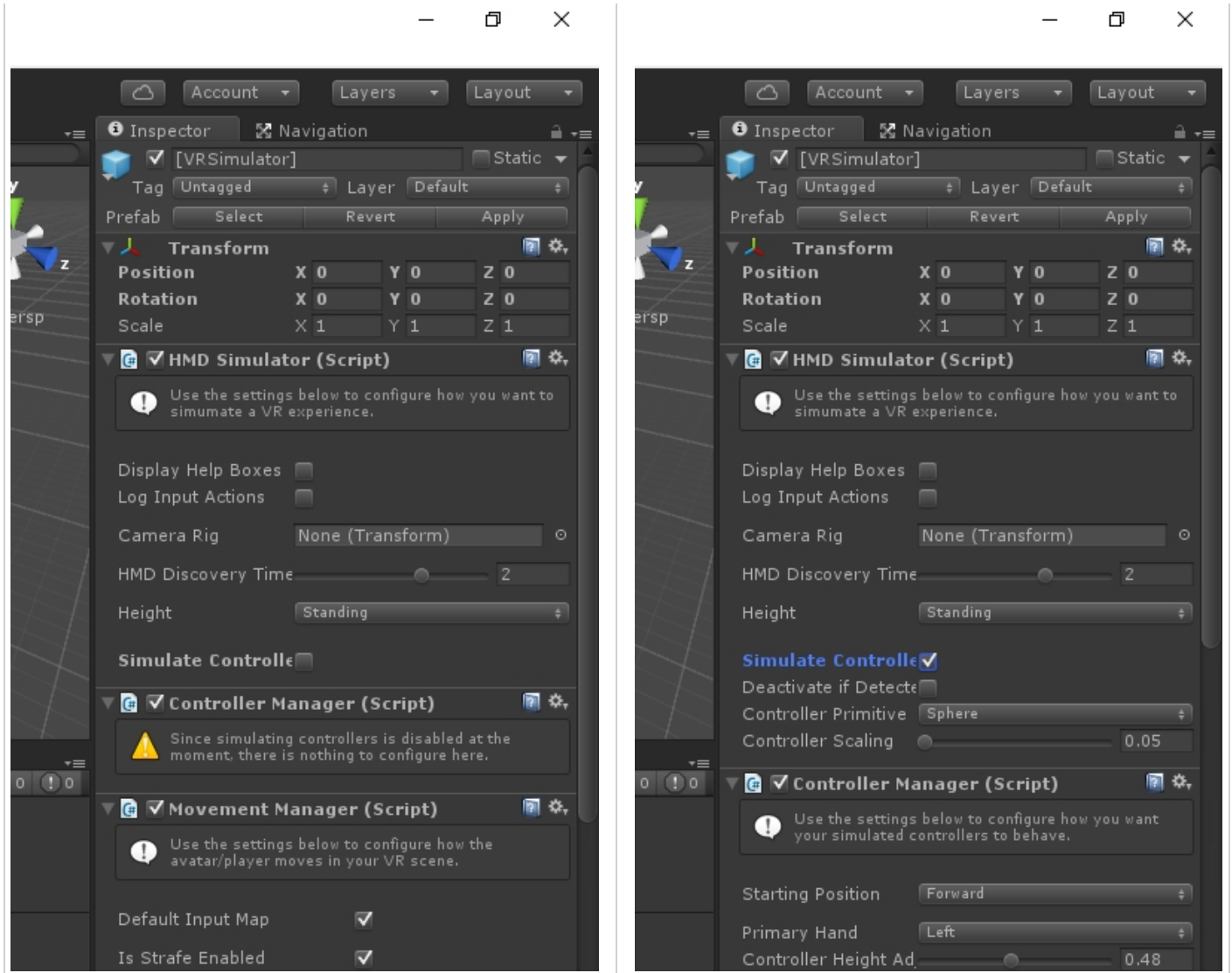
By design, the **VR Simulator** does nothing to compensate for this distortion - it just uses the VR camera rigs as they are, and positions its simulated controllers in worldspace so that they would "look right" rendered in your VR HMD.

This is an intentional design choice, meant to maintain positional, rotational, and scaling consistency between your simulated VR environment and your actual VR environment. If we were to position the simulated controllers so that they seemed undistorted on a regular monitor, then they would not interact with the rest of your VR scene the way your player's actual VR controllers would.

Sorry about the distortion - but the consistency in design and positioning for your actual VR experience is really worth it.

Enabling Controller Simulation

By default, the **VR Simulator** does not simulate position-tracked controllers. To simulate controllers, you need to turn this feature on. You can do so by modifying the **HMD Simulator's Settings (Section 3.3.1)** in the Unity Inspector. Simply make sure that **Simulate Controllers** is enabled, and automatically the VR Simulator will simulate controllers whenever you enter Play Mode.



When **Simulate Controllers** is enabled, new configuration settings will appear in both the **HMD Simulator (Section 3.3.1)** and **Controller Manager (Section 3.3.2)** scripts.

Be Aware

Testing has shown that the SteamVR plugin will only enable position-tracked controllers when an HMD is detected, so it is not currently possible to test real controllers with a simulated HMD.

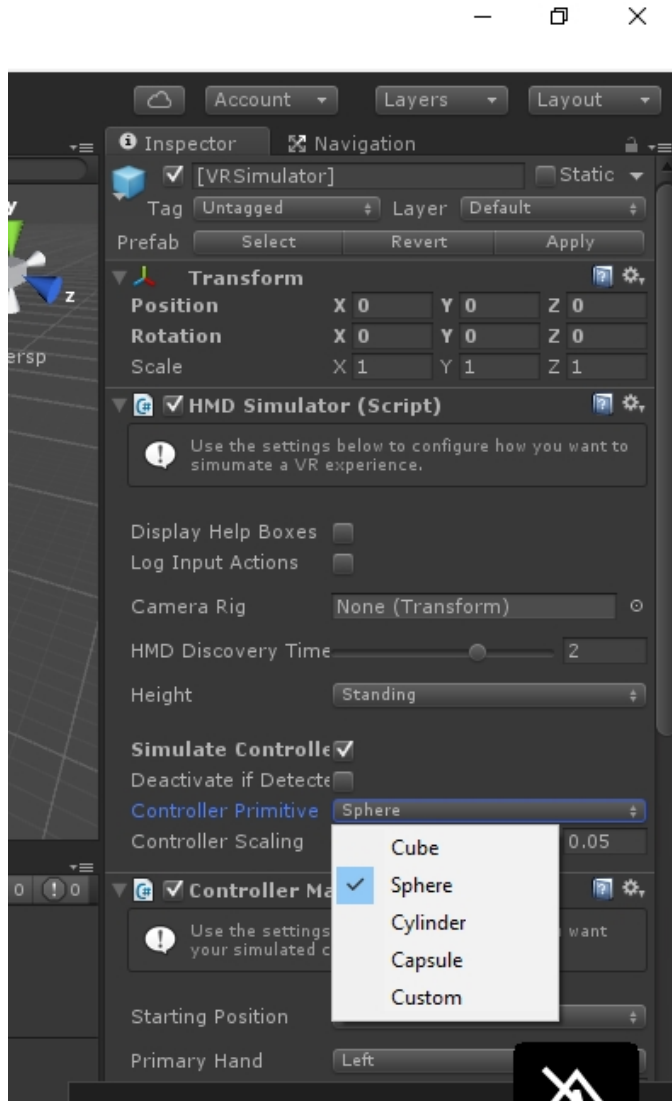
Designing Your Simulated Controllers

When the VR Simulator generates your simulated controllers, it can either create them as:

- A standard [Unity Primitive](#), or;
- A custom prefab.

This is configured in the **HMD Simulator (Section 3.3.1)**'s **Controller Primitive** setting, where you can choose from:

- Cube
- Sphere (default)
- Cylinder
- Capsule
- Custom



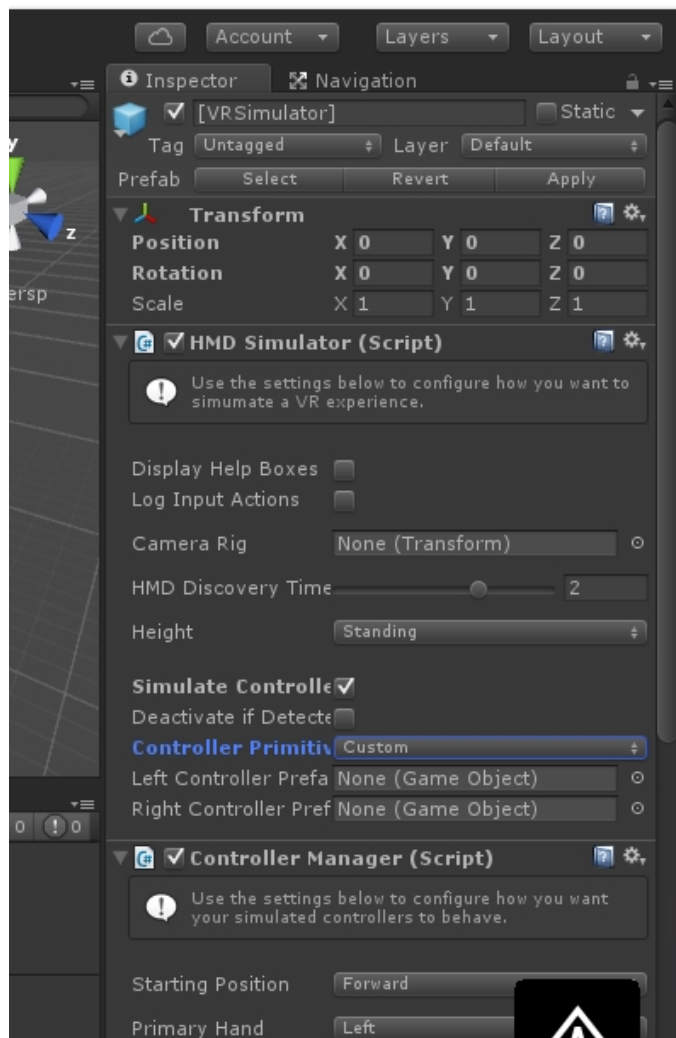
The basic shapes are all standard Unity Primitive Types. Their default scale is (1, 1, 1), but you can scale them down to more-closely resemble actual hands or hand-sized controllers. To do that, just adjust the **Controller Scaling** setting. By default, this setting is set to 0.05 (thus reducing their size by 95%).



Be Aware

If using a standard primitive, both the primitive and its scaling are applied to both controllers. If you want different simulated controllers for the player's right or left hand, then you must use a Custom **Controller Primitive**.


If you set the **Controller Primitive** to **Custom**, you can also specify the prefab to use for your **Left Controller** and **Right Controller**. You can use any prefab from your project's Assets folder - just drag the prefab into the **Left Controller** and **Right Controller** setting, as needed.



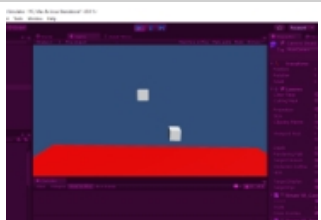


You will note that you cannot adjust the **Controller Scaling** setting for **Custom Controller Primitives**. That is because the simulated controllers will automatically apply whatever scaling their prefab already has configured.

Setting Your Simulated Controllers' Initial Positions

You can set your simulated controllers' initial positions in the **ControllerManager**'s settings in the Unity Inspector. Adjust the **Starting Position** selector to change their initial position. Your options are:

Origin	<p>Simulates "dangling at your waist". They start at more-or-less waist height (lower if seated) and slightly forward of your player avatar's implied body. This slight forward simulation is because players rarely have controllers at rest when in a VR experience, and this lets you more accurately see where your controllers are when in simulation mode.</p>	
---------------	--	---

Forward (default)	Simulates "resting on your desk/keyboard". They are at about torso-height and more substantially forward.	
Reaching	Simulates hands extended out from the shoulder.	
Boxer	Simulates a "puglist's pose" with the hands near eye-level, with one (the primary) hand higher and more forward than the secondary hand.	
Prefab	Uses whatever position automatically derives from the simulated controller's primitive (useful if you're using a custom prefab to simulate a controller).	

You can also configure which of your two controllers should be considered the player's **Primary Hand**. Your options here are simply **Left**, **Right**, or **Ambidextrous** (for both controllers). This setting affects how your hands are positioned if selecting **Boxer** pose.



Best Practice

Depending on the design of your VR experience, you may want the user's left-hand controller to do different things from the right-hand controller.

Even if you aren't using the **Boxer** pose, the **Primary Hand** setting, whose value the **ControllerManager** ('**ControllerManager Class**' in the on-line documentation) class exposes as a public property, can be useful when building and testing such differentiated controller logic.

For More Information...

- **Scripting Reference: ControllerManager** ('**ControllerManager Class**' in the on-line documentation)

By adjusting the **Controller Height Adjustment** you can further tweak how the controllers' are initially positioned. This is particularly useful if you are also using a **Custom Height** for your simulated camera.

Configuring Controller Behavior

Once your simulated controllers are created, they will follow your camera rig as it moves around. If you want them to move independently of the camera rig (for example, if you want to simulate the user actually doing something with their controllers) you can do so by calling **ControllerManager.moveController** ('**moveController Method**' in the on-line documentation) method.

The speed with which the simulated controllers move is determined by the **ControllerManager** (Section 3.3.2)'s **Movement Speed** setting.

Moving Controllers

To move your simulated controllers independently of the camera (i.e. to move a hand, but not the body) you need to programmatically call

the `ControllerManager.moveController` ('moveController Method' in the on-line documentation) method.

This does require scripting, but it is actually very simple. This method takes two forms:

By Position

```
// Moves the left-hand controller from wherever it is to the Forward position.
ControllerManager.moveController(0, ControllerPositions.Forward);

// Moves the right-hand controller from wherever it is to the Origin position.
ControllerManager.moveController(1, ControllerPositions.Origin);
```

As the example above shows, you can simply tell the **ControllerManager** to move the controller to one of the defined positions (which are all relative to the camera/user's avatar). However, if you want to move the controller somewhere else (anywhere else), you can also do so using Unity's standard coordinate system as in the examples below:

To Coordinates

```
// Moves the left-hand controller to the position at (-3, 12, 3) (randomly chosen
// numbers)
// but leaves its rotation the same.
ControllerManager.moveController(0, Vector3(-3, 12, 3),
HMDSimulator.leftController.rotation);

// Moves the right-hand controller to the position at (1, 1.5f, 1) with a new rotation
// from a Quaternion value named "myNewRotation".
ControllerManager.moveController(0, Vector3(1, 1.5f, 1), myNewRotation);
```

3.3 Configuration Settings

3.3.1 HMDSimulator Settings

The **HMDSimulator** script exposes the following settings which are configurable within the Unity Editor:

Display Help Boxes

If enabled, will display helpful explanations in the Unity Editor for all VRSimulator scripts (disabled by default).

Log Input Actions

If enabled, will log any Input Action Event to the Unity Console (disabled by default).

Camera Rig

The Camera Rig that you wish to use for simulation. If left empty (default), will use whichever Camera Rig is present in your scene - if more than one is present, will default to **SteamVR**: [CameraRig].

HMD Discovery Time

Neither the [SteamVR plugin](#) nor [Oculus Utilities for Unity](#) discover a connected HMD right away. It always takes a little time - maybe a few milliseconds, maybe a second or two - depending on your motherboard, graphics card, the resources on your system, etc. This setting lets you configure how many seconds to wait for an HMD before giving up and simulating the HMD. (default: 2 seconds)



Be Aware

On our testing rigs, we've found that an HMD will typically be discovered in about 1.2 seconds if it's connected. But as this has varied widely (even on the same machine), we've found that 2 seconds (our default value) works well as a good compromise. **Your mileage may vary!**

Height

This is the height / position that you wish to simulate. Your options are:

- **Standing (default)** - uses User Profile height if available, otherwise defaults to 1.755 (which for Oculus sets eye-height at approximately 1.65).
- **Seated** - uses User Profile data if available, otherwise defaults 1.06.
- **Custom** - If selected, you can enter your own targeted height at whatever value you choose.

Simulate Controllers

If enabled, will simulate controllers if no HMD is detected. (default: disabled)

Controller Primitive

This is the Unity [primitive](#) type that you wish to use when simulating controllers. Your options are:

- **Cube**
- **Sphere** (default)
- **Cylinder**
- **Capsule**
- **Custom** - If selected, you can select any prefab from your Assets folder. You can use different prefabs for left-hand right controllers if you want.

Controller Scaling

Determines the size of your simulated controller. Assumes each primitive has a size of (1, 1, 1) and scales them

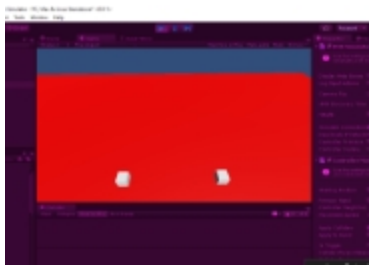
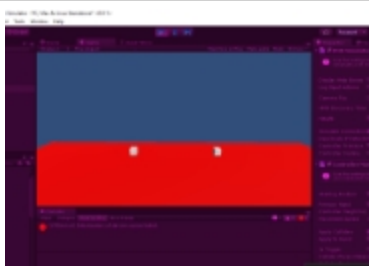
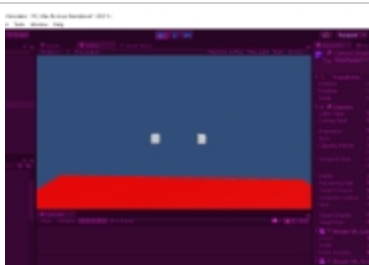

up or down appropriate. If using a custom controller primitive, scaling is determined by the prefab itself. (default: 0.05)

3.3.2 ControllerManager Settings

The **ControllerManager** script exposes the following settings which are configurable within the Unity Editor:

Starting Position

This is the position / posture in which your simulated controllers will start. Your options are:

Origin	Simulates "dangling at your waist". They start at more-or-less waist height (lower if seated) and slightly forward of your player avatar's implied body. This slight forward simulation is because players rarely have controllers at rest when in a VR experience, and this lets you more accurately see where your controllers are when in simulation mode.	
Forward (default)	Simulates "resting on your desk/keyboard". They are at about torso-height and more substantially forward.	
Reaching	Simulates hands extended out from the shoulder.	
Boxer	Simulates a "puglist's pose" with the hands near eye-level, with one (the primary) hand higher and more forward than the secondary hand.	
Prefab	Uses whatever position automatically derives from the simulated controller's primitive (useful if you're using a custom prefab to simulate a controller).	



Be Aware

When using **Play** mode in Unity, your simulated controllers might seem really small or really far away. That's an artifact of how the SteamVR plugin and Oculus Utilities for Unity compositors render your VR scene to a regular (non-HMD) monitor. Basically, you're getting some "lens warping" effects when going from a stereoscopic view (in your HMD) to a non-stereoscopic view (on your monitor).

By design, the VR Simulator does not adjust for that so as to maintain positional, rotational, and scaling consistency with your actual VR scene. In other words, so long as you're positioning things relative to your simulated controllers, they should be positioned "properly" relative to your user's controllers when you're actually wearing an HMD.

Primary Hand

This determines which hand is positioned the furthest-forward when in **Boxer** position. It is also exposed programmatically, letting you give your player "handedness". Your options here are:

- **Left** (default)
- **Right**
- **Ambidextrous** - both hands are treated the same

Controller Height Adjustment

This determines the distance by which the controllers should be offset relative to head height. This is a "magic number" used in downstream calculations - adjust with care! (default: 0.48)

Movement Speed

This determines the rate at which simulated controllers move when moving independently of the user's body. (default: 0.5)

Apply Colliders

If enabled, applies a [Collider](#) to the controller selected by **Apply to Hand**. (default: disabled)

You Should Know...

If **Apply Colliders** is enabled, you can set the same [Collider](#) properties as you would if you were attaching a [Collider](#) component to a [GameObject](#). These properties will appear right below the "**Apply Colliders**" setting, and then be applied to whichever hand is indicated by the **Apply to Hand** setting.

Be Careful!

If you are using a Custom controller primitive, **Apply Colliders** and related settings will **override** any [colliders](#) that have been attached to your custom controller prefab.

Apply to Hand

Indicates which simulated controllers should get the [colliders](#). Your options are:

- **Left** - Only the left-hand controller will get a collider.
- **Right** - Only the right-hand controller will get a collider.
- **Ambidextrous (default)** - Both controllers will get a collider (with identical settings).

Apply Rigidbody

If enabled, applies a [Rigidbody](#) to the controller selected by **Apply to Hand**. (default: disabled)

You Should Know...

If **Apply Rigidbody** is enabled, you can set the same [Rigidbody](#) properties as you would if you were attaching a [Rigidbody](#) component to a [GameObject](#). These properties will appear right below the "**Apply Rigidbody**" setting, and then be applied to whichever hand is indicated by the **Apply to Hand** setting.

Be Careful!

If you are using a Custom controller primitive, **Apply Rigidbody** and related settings will **override** any [rigidbodies](#) that have been attached to your custom controller prefab.

Apply to Hand

Indicates which simulated controllers should get the [colliders](#). Your options are:

- **Left** - Only the left-hand controller will get a collider.
- **Right** - Only the right-hand controller will get a collider.
- **Ambidextrous (default)** - Both controllers will get a collider (with identical settings).

3.3.3 MovementManager Settings

The **MovementManager** script exposes the following settings which are configurable within the Unity Editor:

Default Input Map

If enabled, uses Immerseum's **default movement mapping (Section 4.3)**. (default: enabled)

Be Careful!

If you disable the **Default Input Map** you will have to provide your own movement mapping through scripting.

For More Information...

- **Custom Input Handling (Section 4.1)**

Is Strafe Enabled

If enabled, movement using an input device's x-axis moves the player/user laterally (side-to-side / strafe). If disabled, movement using an input device's x-axis rotate's the camera from side-to-side (yaw). (default: enabled)

Is Run Enabled

If enabled, the player can run (move at twice the default speed). (default: true)

Is Run Active

If enabled, the player is currently running (movement speed is effectively doubled). (default: false)

Gamepad Primary Trigger

This determines which Gamepad Trigger is considered the "primary" one (by default, primary and secondary triggers are mapped differently and so can do different things). Your options are:

- **Left**
- **Right** (default)

Advanced Settings

Be Careful!

The following are advanced settings that have a significant impact on user experience. Adjust them with caution.

Rotation Ratchet

Sets the number of degrees to rotate the user when a ratchet-rotation input action occurs. (default: 45)

Acceleration Rate

Sets the rate at which the user accelerates when starting movement. (default: 0.1)

Forward Damping Rate

Sets the rate by which forward movement is dampened/smoothed. (default: 0.3)

Back/Side Damping Rate

Sets the rate by which backwards and lateral is dampened/smoothed. (default: 0.3)

Gravity Adjustment

Sets the adjustment that occurs to gravity while the player is in motion. (default: 0.379)

3.3.4 InputActionManager Settings

The **InputActionManager** script exposes the following settings which are configurable within the Unity Editor:

Use Immerseum Defaults

If enabled, applies Immerseum's **default Input Actions (Section 4.4)**. This determines which combination of input buttons, axes, etc. on your different input devices do the same things.

Be Careful!

If you disable Immerseum's default Input Actions but still want to use the VR Simulator's movement management system, you will need to supply your own custom Input Actions. This will require scripting.

For More Information...

- **Default Input Actions (Section 4.4)**
- **Custom InputAction Mapping (Section 4.2)**
- **Configuring Input & Movement (Section 4)**

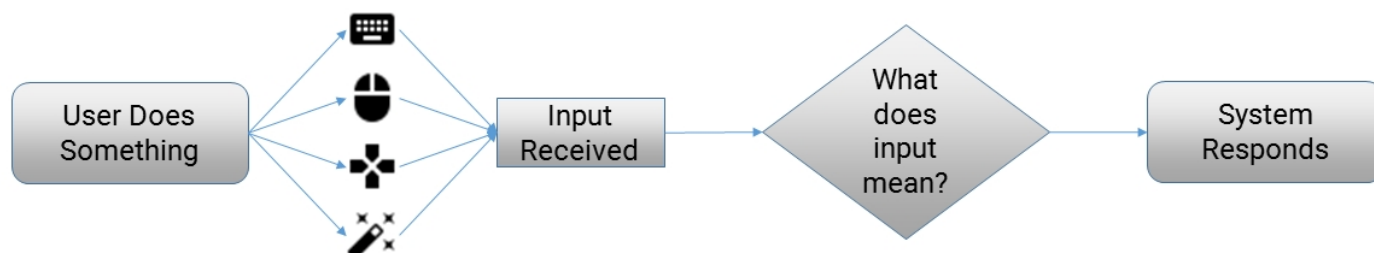
4 Configuring Input & Movement

Gracefully handling input from various user devices and translating that to player movement in your VR scene is the most complicated part of the **VRSimulator**'s functionality. But the good news is that it works right out of the box!

If you want to understand how it works and how to configure it to work with your VR experience, it would be helpful to first explain how input and movement in the **VRSimulator** work.

Overview of Input & Movement

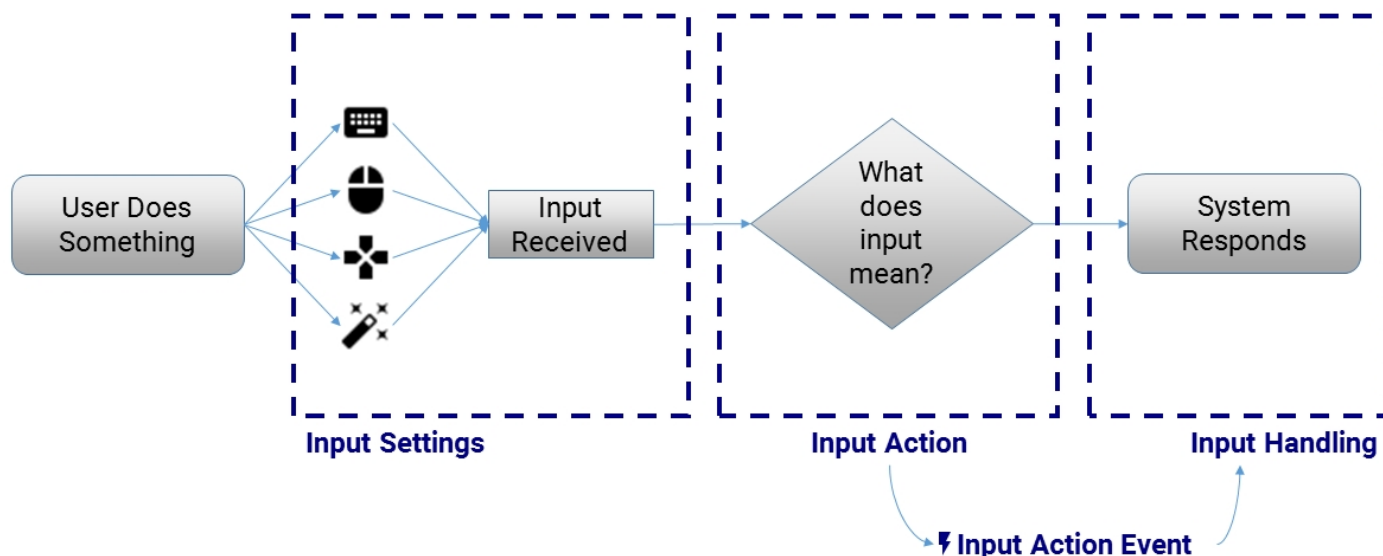
The way movement works in any VR experience (or computer game, etc.) is really straightforward:



The software needs to receive (capture) the user's input from whatever device was used. Based on the input received (what button was pressed, what thumbstick was moved, etc.) the system needs to decide what that input means ("Move the player forward", "Fire the blaster cannon", etc.). And once that decision is made, the system needs to respond and do whatever it should do ("Move the player one step forward", "Start the process of firing the blaster cannon").

Where this can get complicated is that most input devices are built and designed differently, have different hardware features, different drivers, etc. And different experiences (and even different users within the same experience!) may want to have different button layouts, controller configurations, etc. for their style of play. Trying to accommodate all of the variability that's out there can get very complicated, very fast.

Which is why we have broken this process into four key concepts:



- **Input Settings** are the really granular definitions of all of your input sources - buttons, axes, devices, etc. Think of them as the definition of as "where your input comes from". The **VRSimulator** includes a comprehensive set of input settings for standard keyboard, mouse, joystick, and the XBox One controller. When you install the **VRSimulator** plugin these input settings will be set up for your VR scene in the [Unity Input Manager](#).



Be Aware

Not all input devices supported by the VRSimulator are reflected in the [Unity Input Manager](#).

In particular, VR-specific devices like the HTC/Vive Wands, the Oculus Remote, and the Oculus Touch are only accessible via their corresponding Unity plugin ([SteamVR plugin](#) and [Oculus Utilities for Unity](#)).

The **VRSimulator** supports these devices through special device-specific logic which adds an easier-to-understand abstraction layer to the platform-specific APIs.

For More Information...

- [Custom InputAction Mapping \(Section 4.2\)](#)

- **Input Actions** are a collection of inputs that together communicate the same intent. For example, when a user wants to move forward, they can communicate that intent using a variety of inputs: They can push up on a gamepad's right thumbstick, they can press the W key on their keyboard, they can push up on the d-pad, etc. An Input Action collects all of the different inputs that "do the same thing". Then, the **VRSimulator** checks every frame to see which **Input Actions** (if any) have been activated.
- **Input Action Event** is a C# event that gets fired whenever an **Input Action** (any Input Action) occurs. The **Input Action Event** is a programmatic way for the VRSimulator to tell anyone listening for that event "Hey, the user just signalled a desire to do X".
- **Input Handling** is the C# logic that listens for an **Input Action Event**, and then responds accordingly. This process usually consists of two steps: First, identifying the Input Action that occurred (by checking its **name** value), and then calling a method to "do" whatever it is that should be done for that **Input Action**.

So why does any of this matter to you? Honestly, it doesn't need to. The **VRSimulator** comes pre-packaged and configured to include:

- Input settings from keyboard, mouse, XBox One Controller, HTC/Vive Wands, Oculus Remote, and Oculus Touch.
- Input Actions and Input Handling for movement using any supported input devices,
- Input Actions and Input Handling for camera control using any supported input devices,
- Input Actions and Input Handling for other useful actions (e.g. trigger pulls).

If you are comfortable with the way we've defined our **default movement mapping (Section 4.3)**, you don't need to do anything other than configure your user experience in the [Unity Inspector](#).

Configuring Movement

If you are using the VRSimulator's default movement mapping, you can adjust your user experience using the **MovementManager (Section 3.3.3)**'s settings in the Unity Inspector.

Managing Strafe / Look Controls

Strafing is a mainstay in first-person shooter movement, and the VR Simulator supports it. However, strafing is also known to significantly contribute to simulation sickness in VR experiences. Therefore, we've made it an optional feature.

If **Is Strafe Enabled** is activated, then when you press the sideways buttons or move a thumbstick to the side, your player avatar will strafe: They will move sideways in the direction you indicated. To rotate the camera and simulate where your user is looking, you will need to use the default look-only controls on the keyboard, mouse, or gamepad.

However, if **Is Strafe Enabled** is deactivated, then when you press the sideways buttons or move a thumbstick to the side, you will only adjust the direction in which your player avatar is facing (the character's yaw rotation). Your default look-only controls will still work, but now the character can only move forwards / backwards relative to the direction they are facing.

Best Practice

Keeping **Is Strafe Enabled** deactivated makes the simulated experience most-closely match a real VR experience (it's actually surprisingly difficult / weird / rare to strafe in the physical world). That's why it is deactivated by default.

However, there are times when you want to move your user into a particular position to test particular things, and that becomes easier in the simulation if you have strafe enabled (i.e. just slide the character somewhere, then rotate them).

While in **Play Mode** you can activate / deactivate **Is Strafe Enabled** on-the-fly to adjust the UX however you need to.

Managing Run

You can enable or disable the ability for your user to run (move at twice normal speed) using the **Is Run Enabled** setting in the **MovementManager (Section 3.3.3)**. If running is enabled, the default inputs feature controls that toggle run-mode on / off as you play. If running is disabled, these input controls will not make your player run.

You can adjust whether the player is currently in run-mode by toggling the **Is Run Active** setting in the **MovementManager**. This setting is only available if running is enabled, but it will always indicate whether the user is currently in walk (unchecked) or run (checked) mode.

Best Practice

We have found that keeping **Is Run Enabled** activated during the development process to be very helpful.

Often while developing a VR scene we will want to move the user's avatar to a particular position when testing in **Play Mode**. By enabling run mode and then activating it (using the **Is Run Active** setting), we can move the player's avatar through the VR scene very quickly, without messing with any stuff in the inspector that might break our scene.

Managing Your Gamepad Triggers

The Xbox One Controller - like all gamepads - features multiple triggers. These triggers are mapped to two specific InputActions: one for the primary trigger, and one for the secondary trigger. But in order to properly map these InputActions back to the triggers, you need to tell the VRSimulator which trigger you want to be primary (the other will automatically become secondary).

You can do this by selecting Left or Right in the **Gamepad Primary Trigger** setting in the **MovementManager (Section 3.3.3)**.



Be Aware

The **VRSimulator** doesn't have any input handling for either a primary trigger pull or a secondary trigger pull. They're usually not related to movement, after all.

However, by default trigger pulls **are** captured as InputActions so that you can listen for a corresponding Input Action Event and respond accordingly.

For More Information...

- [Custom Input Handling \(Section 4.1\)](#)
- [Default Input Actions \(Section 4.4\)](#)

Customizing Movement Controls

If you do not want to use our **default movement controls (Section 4.3)**, you can **customize input handling (Section 4.1)** to create your own movement logic. **This does not mean you need to define your own custom InputAction mapping (Section 4.2).**

To customize movement controls while using our default InputAction mapping, you would need to:

1. Turn off the **Default Input Map** in the **MovementManager (Section 3.3.3)**'s settings.
2. Write a custom input handling method to respond to each of the **default InputActions (Section 4.4)** that you wish to support.
3. Create an InputAction Event listener method.
4. Check the name of the InputAction against the list of **default Input Actions (Section 4.4)**, and then call the corresponding custom input handling method you wrote in (2) above.

For a walkthrough of how to do this, please see **Custom Input Handling (Section 4.1)**.

Customizing InputActions

Using C# scripting, it is possible to either:

- Modify our default InputActions,
- Extend our default InputActions with new InputActions of your own, or;
- Replace our default InputActions with custom InputActions of your own.

Be Careful!

This is an advanced technique, and one that can get quite complicated. **Use with caution!**

To learn how to customize InputActions, please review **Custom InputAction Mapping (Section 4.2)**.

4.1 Custom Input Handling

While the **VRSimulator** handles most basic movement for you out of the box, there are interactions specific to your VR experience that (obviously) we can't foresee. There might be many situations in which you want your VR experience to react to your user's actions (inputs) beyond simple movement:

- You want your user to pick up an object when their simulated controller is nearby and the primary trigger gets pulled.
- You want to emit a sparkling trail of fire behind your user's avatar when they walk around.
- You want a footstep sound to play when the user steps forward or backward.
- etc.

You can do all of this and more using **Custom Input Handling**, which lets your code listen for our Input Action Events and then respond accordingly.

Important Things to Know

- **Always practice good memory management.** Be sure to both "listen" and "un-listen" to the **EventManager.InputActionStart** ('**OnInputActionStart Event**' in the **on-line documentation**) event in your classes' **OnEnable** and **OnDisable** methods to avoid memory leaks (see below).
- If **Default Input Map** is enabled in the **MovementManager**'s settings, any custom input handling you create that listens for any of our **default InputActions (Section 4.4)** will be additive to our existing input handling. This means you can write a custom input handling method to play a "step" sound, and not worry that your logic will somehow interfere with the user stepping forward.

! Be Careful!

If your custom input handling logic directly affects the player's or camera's position, movement, or rotation it **might** lead to some unexpected results.

Basically, both our default movement / look logic and your custom input handling logic will execute - and we can't guarantee what the final effect will look like.

Listening for an InputAction Event

The key to custom input handling is for your methods to "listen" for an InputAction Event. A method that is listening for (subscribed to) an InputAction Event will always execute whenever that InputAction Event is fired. Below is a sample of a custom InputAction listener method:

myCustomListener

```
public class myCustomClass : MonoBehaviour {  
    void OnEnable() {  
        // "Listen" for the Input Action.  
        EventManager.OnInputActionStart += myCustomListener;  
    }  
}
```

```
}

void OnDisable() {
    // "Unlisten" for the Input Action.
    EventManager.OnInputActionStart -= myCustomListener;
}

void myCustomListener(InputAction firedAction) {
    if (firedAction.name == "DesiredInputAction") {
        // Do something (or call another method) here.
    }
}
}
```

In the code above, the **OnEnable()** method is where you set `myCustomListener` to "listen" for the event **EventManager.OnInputActionStart** ('OnInputActionStart Event' in the on-line documentation) to occur. Essentially, what you're doing is adding the method **myCustomListener** to a list of methods that will be called whenever **OnInputActionStart** occurs.

✓ Best Practice

To make our code easier to follow / keep straight, we always name our listener methods the same as the event they're listening to.

Thus, we wouldn't actually use **myCustomListener** but instead **OnInputActionStart**. Of course, you can use whatever naming conventions you like, but this one works well for us.

The **OnDisable()** method is where you then remove the `myCustomListener` method from the list of methods listening for **OnInputActionStart**. This is to prevent memory leaks - if you didn't "unlisten", then even when you closed Unity, shut down your game, and went off to get a cup of coffee there would still be a lonely little bit of software somewhere waiting in vain for the **OnInputActionStart** event to occur. That's a textbook definition of a nasty memory leak problem.

Reacting to the InputAction

The sample code above contains a method called **myCustomListener** which is a good example of reacting to the Input Action. Here is that code again, just so we can go over its important parts:

myCustomListener

```
void myCustomListener(InputAction firedAction) {
    if (firedAction.name == "DesiredInputAction") {
        // Do something (or call another method) here.
    }
}
```

First, please notice the return type and parameters for the method. **Every single** method that listens for the

`OnInputActionStart` event **must** have the same return type (`void`) and the same parameters (a single **InputAction** ('InputAction Class' in the on-line documentation) object).

! Be Careful!

If your custom listener for the `OnInputActionStart` event does not have a return type of `void` and accept a single **InputAction** ('InputAction Class' in the on-line documentation) parameter, your code **will** throw an exception.

Second, notice the `if { ... }` statement inside of the `myCustomListener` method. This `myCustomListener` method will execute whenever **any** **InputAction** is detected, regardless of what that **InputAction** actually represents. But presumably, you want your `myCustomListener` to only respond to a particular **InputAction** ('InputAction Class' in the on-line documentation) (a primary trigger pull, for example). This `if { ... }` statement makes sure your code is only executed for the specific **InputAction** ('InputAction Class' in the on-line documentation) (s) that you want - in the example above, an **InputAction** ('InputAction Class' in the on-line documentation) named "DesiredInputAction".

And that's it! There you have a custom input handler.

✓ Best Practice

If you're using our default **InputActions** but wish to create custom handlers for them, there is an easier way: Each of our default **InputActions** has its own **InputAction**-specific events which get fired whenever the **InputAction** is detected.

This allows you to listen for a specific **InputAction** without checking its **name** property as shown above.

For more information on the events fired by our default **Input Actions**, please see the **Default Input Actions (Section 4.4)** reference.

For More Information...

- **Default Input Actions (Section 4.4)**
- **Scripting Reference: Event Reference (Section 5.2)**
- **Scripting Reference: EventManager (Section 4.4)**

Customizing Movement

There are three ways to customize the `VRSimulator`'s movement logic:

1. Retain our **default Input Actions (Section 4.4)** and just replace our movement logic,
2. Use **your own Input Actions (Section 4.2)**, but keep Immerseum's default movement logic,
3. Use **your own Input Actions (Section 4.2)** and your own movement logic.

Retaining Immerseum Input Actions with Custom Movement

If you wish to retain our **default Input Actions (Section 4.4)** while customizing movement, you need to do the following:

1. Turn off **Default Input Map** in the **MovementManager (Section 3.3.3)**'s settings.
2. Write a custom listener for each of the default Input Actions you wish to support.

The best (easiest to understand and most-performative) way to do this is to use one "switchboard" method as in the example below:

Custom Switchboard Listener

```
void OnEnable() {
    EventManager.OnInputActionStart += myCustomSwitchboard;
}

void OnDisable() {
    EventManager.OnInputActionStart -= myCustomSwitchboard;
}

void myCustomSwitchboard(InputAction firedAction) {
    switch (firedAction.name) {
        case "togglePauseMenu":
            // ... Custom Code Here
            break;
        case "toggleView":
            // ... Custom Code Here
            break;
        case "togglePrimaryTrigger":
            // ... Custom Code Here
            break;
        case "toggleSecondaryTrigger":
            // ... Custom Code Here
            break;
        case "toggleRightThumbstickClick":
            // ... Custom Code Here
            break;
        case "toggleLeftThumbstickClick":
            // ... Custom Code Here
            break;
        case "toggleRightBumper":
            // ... Custom Code Here
            break;
        case "toggleLeftBumper":
            // ... Custom Code Here
            break;
        case "toggleSelectionButton":
            // ... Custom Code Here
            break;
        case "toggleCancelButton":
            // ... Custom Code Here
            break;
    }
}
```

```
case "toggleSecondaryButton":
    // ... Custom Code Here
    break;
case "toggleTertiaryButton":
    // ... Custom Code Here
    break;
case "xAxisMovement":
    // ... Custom Code Here
    break;
case "zAxisMovement":
    // ... Custom Code Here
    break;
case "pitchRotation":
    // ... Custom Code Here
    break;
case "yawRotation":
    // ... Custom Code Here
    break;
}
```

Given the code above, just replace the comments `// ... Custom Code Here` with your own method calls, and you'll have successfully rewired our movement system.

Using Custom Input Actions with Default Movement

If you wish to use your own custom Input Actions but retain all of Immerseum's movement logic, you should:

1. Turn off **Use Immerseum Defaults** in the **InputActionManager (Section 3.3.4)**'s settings.
2. Define your own **custom InputActions (Section 4.2)** with the same names as our default InputActions (**Section 4.4**).
3. At runtime, register your custom InputActions with the **InputActionManager ('InputActionManager Class' in the on-line documentation)** as described in **Custom InputAction Mapping (Section 4.2)** when the **EventManager.OnInitializeInputActions ('OnInitializeInputActions Event' in the on-line documentation)** fires.

And that's it! Now the **VRSimulator** movement system will move the user in response to your **custom InputActions (Section 4.2)**. For more information, please see **Replacing Default InputActions (Section 4.2)**.

Using Custom Input Actions with Custom Movement

If you wish to use your own custom Input Actions with your own custom movement logic, you should:

1. Turn off **Default Input Map** in the **MovementManager (Section 3.3.3)**'s settings.
2. Turn off **Use Immerseum Defaults** in the **InputActionManager (Section 3.3.4)**'s settings.
3. Define your own **custom InputActions (Section 4.2)**. In this case, you can use whatever names you want.
4. Write a **custom listener** or a **switchboard listener** for each of the custom InputActions you wish to support.
5. At runtime, register your custom InputActions with the **InputActionManager ('InputActionManager Class' in the on-line documentation)** as described in **Custom InputAction Mapping (Section 4.2)** when the **EventManager.OnInitializeInputActions ('OnInitializeInputActions Event' in the**

`on-line documentation)` fires.

And that's it! Your InputActions will now result in whatever movement you've defined through your custom listeners.

4.2 Custom InputAction Mapping

While the **VRSimulator** has defined a rather comprehensive set of **default Input Actions (Section 4.4)** for you, there may be situations where you wish to either:

- modify our default Input Actions,
- extend our default Input Actions with new (additional) Input Actions of your own, or;
- replace our Input Actions with your own.

Using C# scripting and the **InputAction** ('InputAction Class' in the on-line documentation), **InputAxis** ('InputAxis Class' in the on-line documentation), and **InputButton** ('InputButton Class' in the on-line documentation) classes (and their derivatives) you can do all three of the items above.

Be Careful!

We consider **Custom InputAction Mapping** to be the most complicated type of integration with the **VRSimulator**.

Because it directly relates to the inputs that will be captured from your users, mistakes, bugs, inefficiencies, etc. made here are likely to fundamentally destabilize your user experience.

Proceed at your own risk.

How InputActions Work

The InputActionManager Explained

Input Actions are coordinated by the **InputActionManager** [[Configuration \(Section 3.3.4\)](#) | [Scripting \('InputActionManager Class' in the on-line documentation\)](#)]. The **InputActionManager** is a singleton class which exposes the majority of its functionality via static methods. Its responsibilities are simple:

- **Store and manage the list of registered InputActions.** These are the InputActions that the platform is waiting to receive.
- **Check for InputActions.** Every frame, check whether one or more of the registered InputActions has been received and if so, fire the **EventManager.OnInputActionStart** ('OnInputActionStart Event' in the on-line documentation) event.

While the **InputActionManager** exposes a variety of static properties and methods that you can read from, you will most likely be using:

- **Properties**
 - **InputActionManager.inputActionList** ('inputActionList Property' in the on-line documentation). This is a [List<InputAction](#) ('InputAction Class' in the on-line documentation) that contains all of the InputActions that have been registered with the **InputActionManager**.
- **Methods**
 - **InputActionManager.addAction(InputAction action)** ('addAction Method' in the on-line documentation). This adds an InputAction to the list of registered Input Actions.
 - **InputActionManager.getInputAction(string name)** ('getInputAction Method' in the

`on-line documentation`). This returns the `InputAction` from the list of registered `InputActions` whose `name` property matches the supplied string.

- `InputActionManager.removeInputAction(string name)` ('removeInputAction Method' in the on-line documentation). This removes the `InputAction` whose `name` property matches the supplied string.
- `InputActionManager.isActionRegistered(string name)` ('isActionRegistered Method' in the on-line documentation). This returns a boolean value indicating whether an `InputAction` whose `name` matches the supplied string is registered or not.

Anatomy of an InputAction

An **InputAction** ('InputAction Class' in the on-line documentation) represents a collection of inputs that correspond to the same intent. For example:

- "move Forward" would be one `InputAction` that defines all of the various ways that the user's intent to move forward can be captured from supported input devices, or;
- "pull primary trigger" would be one `InputAction` that defines all of the various ways that the user has pulled a primary trigger.

While it is possible for an **InputAction** ('InputAction Class' in the on-line documentation) to correspond to one and only one input device's button/axis, the key benefit of using `InputActions` is to simplify code by applying the same logic to all inputs that mean the same thing.

InputActions ('InputAction Class' in the on-line documentation) only expose two public methods: `InputAction.registerAction()` ('registerAction Method' in the on-line documentation) and `InputAction.deregisterAction()` ('deregisterAction Method' in the on-line documentation). As the method names suggest, these methods register and deregister the `InputAction` with the **InputActionManager** ('InputActionManager Class' in the on-line documentation).

The key component of the `InputAction`, however, are its properties. These properties can be divided into three categories:

- **Descriptive.** These properties provide meta-data scaffolding for identifying, describing, and working with the `InputAction`. Some are read-only, whereas some can be modified.
- **Input Definition.** These properties define the inputs that are associated with (fire/invoke) the `InputAction`.
- **Results.** These properties are read-only properties which return the output of an `InputAction` (values determined by the input devices that are used).

The list below describes those properties which are most important for defining custom `InputActions`:

- **name** - this is used to identify the `InputAction`, and must be unique within the scope of the `InputActionManager.inputActionList` ('inputActionList Property' in the on-line documentation).
- **pressedKeyList** - this is a `List<UnityEngine.KeyCode>` of those keys which will invoke the `InputAction` if pressed.
- **releasedKeyList** - this is a `List<UnityEngine.KeyCode>` of those keys which will invoke the `InputAction` if released.
- **heldKeyList** - this is a `List<UnityEngine.KeyCode>` of those keys which will invoke the `InputAction` if held.
- **pressedAndHeldKeyList** - this is a `List<UnityEngine.KeyCode>` of those keys which will invoke the `InputAction` if pressed and then held afterwards.
- **xAxisList** - this is a `List<InputAxis` ('InputAxis Class' in the on-line documentation)> of those `InputAxis` ('InputAxis Class' in the on-line documentation) which are intended to adjust the user's position along the X-

axis (left/right).

- **yAxisList** - this is a [List<InputAxis \('InputAxis Class' in the on-line documentation\)>](#) of those **InputAxis ('InputAxis Class' in the on-line documentation)** which are intended to adjust the user's position along the Y-axis (up/down).
- **zAxisList** - this is a [List<InputAxis \('InputAxis Class' in the on-line documentation\)>](#) of those **InputAxis ('InputAxis Class' in the on-line documentation)** which are intended to adjust the user's position along the Z-axis (forward/backward).
- **rightTriggerAxisList** - this is a [List<InputAxis \('InputAxis Class' in the on-line documentation\)>](#) of those **InputAxis ('InputAxis Class' in the on-line documentation)** which correspond to the input device's right-hand trigger.
- **leftTriggerAxisList** - this is a [List<InputAxis \('InputAxis Class' in the on-line documentation\)>](#) of those **InputAxis ('InputAxis Class' in the on-line documentation)** which correspond to the input device's left-hand trigger.
- **pitchAxisList** - this is a [List<InputAxis \('InputAxis Class' in the on-line documentation\)>](#) of those **InputAxis ('InputAxis Class' in the on-line documentation)** which are intended to adjust the camera's pitch rotation.
- **yawAxisList** - this is a [List<InputAxis \('InputAxis Class' in the on-line documentation\)>](#) of those **InputAxis ('InputAxis Class' in the on-line documentation)** which are intended to adjust the user's position along the X-axis (left/right).
- **pressedButtonList** - this is a [List<InputButton \('InputButton Class' in the on-line documentation\)>](#) of those **InputButtons ('InputButton Class' in the on-line documentation)** which will invoke the InputAction if pressed.
- **releasedKeyList** - this is a [List<InputButton \('InputButton Class' in the on-line documentation\)>](#) of those **InputButtons ('InputButton Class' in the on-line documentation)** which will invoke the InputAction if released.
- **heldKeyList** - this is a [List<InputButton \('InputButton Class' in the on-line documentation\)>](#) of those **InputButtons ('InputButton Class' in the on-line documentation)** which will invoke the InputAction if held.
- **pressedAndHeldKeyList** - this is a [List<InputButton \('InputButton Class' in the on-line documentation\)>](#) of those **InputButtons ('InputButton Class' in the on-line documentation)** which will invoke the InputAction if pressed and then held afterwards.



Best Practice

Many InputActions will feature both keyboard, button, or axis inputs. For example, one potential InputAction would be to "move forward" based either on the keyboard's W key, the gamepad's right thumbstick, or the mouse's scroll wheel.

To properly define such an InputAction, these would need to be included in the appropriate properties described above.

**Be Aware**

Technically, there is nothing stopping you from including a particular input (a [UnityEngine.KeyCode](#), an **InputAxis** ('**InputAxis Class**' in the on-line documentation), or an **InputButton** ('**InputButton Class**' in the on-line documentation)) in multiple **InputActions** ('**InputAction Class**' in the on-line documentation).

This might lead to some strange behavior (because one input will actually be tied to multiple intentions), but there are certain situations where that may be desired behavior.

For example, it would be perfectly reasonable to include the same input in three **InputActions** where one **InputAction** will only apply in the "normal" state, a different one will apply if your user has paused the game, and a third might apply if the user is in a "swimming" state.

Anatomy of an InputAxis

An **InputAxis** ('**InputAxis Class**' in the on-line documentation) is used to define a single input which returns a quantitative value of its state (as opposed to a boolean, like a keyboard input). In a non-VR game, such input axes get defined in the [Unity Input Manager](#) and you can just access them using Unity's brilliant [Input](#) API.

However, one of the challenges in developing for VR is that the [Unity Input Manager](#) does not support VR-compatible input devices like the HTC/Vive Wands and Oculus Touch. They can only be accessed through the [SteamVR](#) and [Oculus Utilities for Unity](#) plugins respectively.

To address this challenge, the VR Simulator input system supports three types of **InputAxis** ('**InputAxis Class**' in the on-line documentation) (all derive from the generic **InputAxis** ('**InputAxis Class**' in the on-line documentation)):

- **InputAxis** ('**InputAxis Class**' in the on-line documentation) - this is the generic **InputAxis** that corresponds to any **InputAxis** defined through the [Unity Input Manager](#).
- **OculusInputAxis** ('**OculusInputAxis Class**' in the on-line documentation) - this is an **InputAxis** that is only accessible through the [Oculus Utilities for Unity](#) plugin.
- **SteamVRInputAxis** ('**SteamVRAxis Class**' in the on-line documentation) - this is an **InputAxis** that is only accessible through the [SteamVR](#) plugin.

Each **InputAxis** has a set of read-only properties which return the results (the value) of the axis at any given moment. They are fairly clearly **documented** ('**InputAxis Class**' in the on-line documentation).

However, when creating a custom **InputAction** it is important to understand the definition fields that need to be configured:

- **name** - This is the name given to the axis.

**Be Careful!**

If defining a generic **InputAxis** ('**InputAxis Class**' in the on-line documentation), it must correspond to the name set for the axis in the [Unity Input Manager](#), otherwise it will not capture the input's state properly.

- **valueAtRestList** - this is a [List<float>](#) of values which if returned by the axis can be considered "resting" (i.e.

not moving). This is a list because some devices can have multiple at-rest positions for particular inputs.

- **hasMinimumValue** - if true, indicates that the axis does have a value below which it will not go.
- **hasMaximumValue** - if true, indicates that the axis does have a maximum value above which it will not go.
- **minimumValue** - the minimum value below which the axis will not go.
- **maximumValue** - the maximum value above which the axis will not go.

An **OculusInputAxis** ('**OculusInputAxis Class**' in the on-line documentation) receives certain additional definition properties:

- **controllerMask** - indicates which Oculus-compatible controllers to capture the axis value from.
- **axis1D** - the bitmask that identifies a 1-dimensional Oculus axis
- **axis2D** - the bitmask that identifies a 2-dimensional Oculus axis

A **SteamVRInputAxis** ('**SteamVRAxis Class**' in the on-line documentation) also receives certain additional definition properties:

- **buttonMask** - the bitmask associated with a button associated with the input axis
- **deviceRelation** - indicates which controller to capture the axis information from.

Anatomy of an Input Button

An **InputButton** ('**InputButton Class**' in the on-line documentation) is used to define a single input which returns a button state (pressed, released, held, etc.). In a non-VR game, such buttons get defined in the [Unity Input Manager](#) and you can just access them using Unity's brilliant [Input](#) API.

However, one of the challenges in developing for VR is that the [Unity Input Manager](#) does not support VR-compatible input devices like the HTC/Vive Wands and Oculus Touch. They can only be accessed through the [SteamVR](#) and [Oculus Utilities for Unity](#) plugins respectively.

To address this challenge, the VR Simulator input system supports multiple types of **InputButton** ('**InputButton Class**' in the on-line documentation) (all derive from the generic **InputButton** ('**InputButton Class**' in the on-line documentation)):

- ('**InputAxis Class**' in the on-line documentation)**InputButton** ('**InputButton Class**' in the on-line documentation) - this is the generic InputButton that corresponds to a gamepad input button defined through the [Unity Input Manager](#).
- **MouseButton** ('**MouseButton Class**' in the on-line documentation) - this is a particular type of InputButton that corresponds to a mouse button defined through the [Unity Input Manager](#).
- **OculusButton** ('**OculusButton Class**' in the on-line documentation) - this is a generic InputButton that is only accessible through the [Oculus Utilities for Unity](#) plugin.
- **OculusTouchButton** ('**OculusTouchButton Class**' in the on-line documentation) - this is a particular type of button that is only accessible through the [Oculus Utilities for Unity](#) plugin.
- **OculusNearTouchButton** ('**OculusNearTouchButton Class**' in the on-line documentation) - this is a particular

type of button that is only accessible through the [Oculus Utilities for Unity](#) plugin.

- **SteamVRButton** ('**SteamVRButton Class**' in the on-line documentation) - this is a generic InputButton that is only accessible through the [SteamVR](#) plugin.
- **SteamVRHairTrigger** ('**SteamVRHairTrigger Class**' in the on-line documentation) - this is a specific button type for the HairTrigger accessible through the [SteamVR](#) plugin.
- **SteamVRTouch** ('**SteamVRTouch Class**' in the on-line documentation) - this is a specific button type for the SteamVR touchpad only accessible through the [SteamVR](#) plugin.

Each InputButton has a set of read-only properties which return the results (the value) of the button at any given moment. They are fairly clearly **documented** ('**InputButton Class**' in the on-line documentation).

However, when creating a custom InputAction it is important to understand the definition fields that need to be configured:

- **name** - This is the name given to the button. It applies to all types of InputButton.

A **MouseButton** ('**MouseButton Class**' in the on-line documentation) receives certain additional definition properties:

- **buttonIndex** - indicates the specific button on the physical mouse.

All **OculusButtons** ('**MouseButton Class**' in the on-line documentation) (including **OculusTouchButton** ('**OculusButton Class**' in the on-line documentation) and **OculusNearTouchButton** ('**OculusNearTouchButton Class**' in the on-line documentation)) receives certain additional definition properties:

- **buttonMask** - provides a bitmask of the the specific button on the physical controller.
- **rawButtonMask** - provides a bitmask of the specific button's raw value source on the physical controller.
- **controllerMask** - indicates which Oculus-compatible controllers to capture the axis value from.

All **SteamVRButtons** ('**SteamVRAxis Class**' in the on-line documentation) (including **SteamVRHairTrigger** ('**SteamVRHairTrigger Class**' in the on-line documentation) and **SteamVRTouch** ('**SteamVRTouch Class**' in the on-line documentation)) also receives certain additional definition properties:

- **buttonMask** - the bitmask associated with the button.
- **deviceRelation** - indicates which controller to capture the button information from.

Modifying Registered InputActions

The best way to modify **default InputActions** (Section 4.4) is to do so when the **EventManager.OnInitializeInputActionsEnd** ('**OnInitializeInputActionsEnd Event**' in the on-line documentation) event fires. At that point, all InputActions have been registered with the **InputActionManager** ('**InputActionManager Class**' in the on-line documentation) and can safely be modified without fear of being over-written by the VR Simulator's initialization logic.

There are a variety of ways to programmatically modify registered InputActions, but the way that we recommend is to:

1. Retrieve the **InputAction** ('**InputAction Class**' in the on-line documentation) from the **InputActionManager** ('**InputActionManager Class**' in the on-line documentation).
2. Make appropriate modifications to the **InputAction** ('**InputAction Class**' in the on-line documentation).

3. Deregister the modified **InputAction** ('InputAction Class' in the on-line documentation) from the **InputActionManager** ('InputActionManager Class' in the on-line documentation).
4. Register the modified **InputAction** ('InputAction Class' in the on-line documentation) from the **InputActionManager** ('InputActionManager Class' in the on-line documentation).



Be Aware

The workflow described above will not work properly if you modify the InputAction's **name** property (this is used to uniquely identify the InputAction, after all).

The following is an example of this workflow in action. This example adds pressing the Spacebar to the "togglePauseMenu" default InputAction:

Modifying Default InputAction

```
using UnityEngine;
using Immerseum.VRSimulator;

public class myClass : MonoBehaviour {
    void OnEnable() {
        EventManager.OnInitializeInputActionsEnd += modifyInputAction;
    }

    void OnDisable() {
        EventManager.OnInitializeInputActionsEnd -= modifyInputAction;
    }

    void modifyInputAction() {
        InputAction actionToModify =
        InputActionManager.getInputAction("togglePauseMenu");
        actionToModify.pressedKeyList.Add(KeyCode.Space);
        actionToModify.deregisterAction();
        actionToModify.registerAction();
    }
}
```

Extending Default InputActions

Adding a new InputAction to the VRSimulator's movement system is relatively simple:

1. Create an **InputAction** ('InputAction Class' in the on-line documentation) with an appropriate and unique name (i.e. a name that doesn't already exist among registered InputActions).
2. Call the **InputAction.registerAction()** ('registerAction Method' in the on-line documentation) method on it.

Below is an example that does this to create a new "PressTheKButton" InputAction:

Extending InputActions

```
using UnityEngine;
using Immerseum.VRSimulator;

public class myClass : MonoBehaviour {
    void Start() {
        InputAction myInputAction = new InputAction();
        myInputAction.name = "pressTheKButton";
        myInputAction.pressedKeyList.Add(KeyCode.K);
        myInputAction.registerAction();
    }
}
```

Replacing Default Input Actions

If you intend to replace all of the **default InputActions (Section 4.4)** with your own, you don't have to go through the modification workflow described above for each one. It is easier to:

1. Turn off **Use Immerseum Defaults** in the **InputActionManager (Section 3.3.4)**'s settings.
2. Define your own **custom InputActions with the same names as our default InputActions (Section 4.4)**.
3. At runtime, register your custom InputActions with the **InputActionManager ('InputActionManager Class' in the on-line documentation)**. This can most easily occur when **EventManager.OnInitializeInputActions** ('**OnInitializeInputActions Event**' in the on-line documentation) fires.

The example below provides some sample code for doing so:

Replacing Default Input Actions

```
using UnityEngine;
using Immerseum.VRSimulator;

public class myClass : MonoBehaviour {
    void OnEnable() {
        EventManager.OnInitializeInputActions += createCustomInputActions;
    }

    void OnDisable() {
        EventManager.OnInitializeInputActions -= createCustomInputActions;
    }

    void createCustomInputActions() {
        InputAction firstInputAction = new InputAction();
        firstInputAction.name = "togglePauseMenu";
        firstInputAction.pressedKeyList.Add(KeyCode.Space);
        firstInputAction.registerAction();

        // Repeat for additional Input Actions
    }
}
```

```
}  
}
```

Provided you left the **Default Input Map** setting enabled in the **MovementManager**'s settings, you will now simply have different buttons activating the same movement logic.

Replacing Default Input Actions and Movement Logic

Putting all of the above together with **Custom Input Handling (Section 4.1)**, it is possible to both replace all of the default Input Actions and apply your own custom movement logic. Simply:

1. Turn off **Default Input Maps** in the **MovementManager (Section 3.3.3)** settings.
2. Follow the steps for **Replacing Default Input Actions** described above.
3. Either create **custom listeners (Section 4.1)** for each of your custom InputActions, or create a **switchboard listener (Section 4.1)** for all of them.

4.3 Default Movement Mapping

By default, the **VRSimulator** supports a relatively standard set of movement inputs. The list below indicates the movement supported on each type of input device:

You Should Know...



The VRSimulator supports more than the default InputActions described below. However, those InputActions merely register that particular inputs (like triggers, buttons, etc.) have been engaged - they do not automatically result in any kind of movement.



If you would like to use those InputActions, you can do so using **Custom Input Handling (Section 4.1)**.

For More Information...

- **Default Input Actions (Section 4.4)**
- **Custom Input Handling (Section 4.1)**

Keyboard

Movement Type	Inputs	Related Default InputAction (Section 4.4)
Move Forward	<ul style="list-style-type: none"> • W • Up Arrow • Numpad 8 	zAxisMovement
Move Backward	<ul style="list-style-type: none"> • S • Down Arrow • Numpad 2 	zAxisMovement
Move Left	<div>  Be Aware If Is Strafe Enabled (Section 3.3.3) is activated </div> <ul style="list-style-type: none"> • A • Left Arrow • Numpad 4 	xAxisMovement
Move Right	<div>  Be Aware </div>	xAxisMovement

Movement Type	Inputs	Related Default InputAction (Section 4.4)
	<p>If Is Strafe Enabled (Section 3.3.3) is activated</p> <ul style="list-style-type: none"> • D • Right Arrow • Numpad 6 	
Look Up		pitchRotation
Look Down		pitchRotation
Look Left	<p> Be Aware</p> <p>If Is Strafe Enabled (Section 3.3.3) is deactivated</p> <ul style="list-style-type: none"> • A • Left Arrow • Numpad 4 	yawRotation
Look Right	<ul style="list-style-type: none"> • D • Right Arrow • Numpad 6 	yawRotation
Left Look Ratchet	<ul style="list-style-type: none"> • Q • Numpad 7 	toggleLeftBumper
Right Look Ratchet	<ul style="list-style-type: none"> • E • Numpad 9 	toggleRightBumper
Toggle Run Mode	<p> Be Aware</p> <p>If Is Run Enabled (Section 3.3.3) is activated</p> <ul style="list-style-type: none"> • C • Left Shift 	toggleLeftThumbstickClick
Center Camera	<ul style="list-style-type: none"> • F 	toggleRightThumbstickClick


Movement Type	Inputs	Related Default InputAction (Section 4.4)
Forward	<ul style="list-style-type: none"> Numpad 5 	

Mouse

Movement Type	Inputs	Related Default InputAction (Section 4.4)
Move Forward		zAxisMovement
Move Backward		zAxisMovement
Move Left		xAxisMovement
Move Right		xAxisMovement
Look Up	<ul style="list-style-type: none"> Vertical (up) 	pitchRotation
Look Down	<ul style="list-style-type: none"> Vertical (down) 	pitchRotation
Look Left	<ul style="list-style-type: none"> Horizontal (left) 	yawRotation
Look Right	<ul style="list-style-type: none"> Horizontal (right) 	yawRotation
Left Look Ratchet		toggleLeftBumper
Right Look Ratchet		toggleRightBumper
Toggle Run Mode		toggleLeftThumbstickClick
Center Camera Forward	<ul style="list-style-type: none"> Center Button 	toggleRightThumbstickClick


Gamepad: Xbox One Controller

Movement Type	Inputs	Related Default InputAction (Section 4.4)
Move Forward	<ul style="list-style-type: none"> Left Thumbstick Y (up) D-Pad Up 	zAxisMovement
Move Backward	<ul style="list-style-type: none"> Left Thumbstick Y (down) 	zAxisMovement
Move Left	<ul style="list-style-type: none"> Left Thumbstick X (left) D-Pad Left 	xAxisMovement

Movement Type	Inputs	Related Default InputAction (Section 4.4)
Move Right	<ul style="list-style-type: none"> Left Thumbstick X (right) D-Pad Right 	xAxisMovement
Look Up	<ul style="list-style-type: none"> Right Thumbstick Y (up) 	pitchRotation
Look Down	<ul style="list-style-type: none"> Right Thumbstick Y (down) 	pitchRotation
Look Left	<ul style="list-style-type: none"> Right Thumbstick X (left) 	yawRotation
Look Right	<ul style="list-style-type: none"> Right Thumbstick X (right) 	yawRotation
Left Look Ratchet	<ul style="list-style-type: none"> Left Bumper 	toggleLeftBumper
Right Look Ratchet	<ul style="list-style-type: none"> Right Bumper 	toggleRightBumper
Toggle Run Mode	<div>  Be Aware If Is Run Enabled (Section 3.3.3) is activated </div> <ul style="list-style-type: none"> Left Thumbstick (click) 	toggleLeftThumbstickClick
Center Camera Forward	<ul style="list-style-type: none"> Right Thumbstick (click) 	toggleRightThumbstickClick

HTC/Vive Wand

Movement Type	Inputs	Related Default InputAction (Section 4.4)
Move Forward		zAxisMovement
Move Backward		zAxisMovement
Move Left		xAxisMovement
Move Right		xAxisMovement
Look Up		pitchRotation
Look Down		pitchRotation
Look Left		yawRotation


Movement Type	Inputs	Related Default InputAction (Section 4.4)
Look Right		yawRotation
Left Look Ratchet		toggleLeftBumper
Right Look Ratchet		toggleRightBumper
Toggle Run Mode	<div>  Be Aware If Is Run Enabled (Section 3.3.3) is activated </div> <ul style="list-style-type: none"> • Left Grip 	toggleLeftThumbstickClick
Center Camera Forward	<ul style="list-style-type: none"> • Right Grip 	toggleRightThumbstickClick

Oculus Remote

Movement Type	Inputs	Related Default InputAction (Section 4.4)
Move Forward	<ul style="list-style-type: none"> • D-Pad Up 	zAxisMovement
Move Backward	<ul style="list-style-type: none"> • D-Pad Down 	zAxisMovement
Move Left	<ul style="list-style-type: none"> • D-Pad Left 	xAxisMovement
Move Right	<ul style="list-style-type: none"> • D-Pad Right 	xAxisMovement
Look Up		pitchRotation
Look Down		pitchRotation
Look Left		yawRotation
Look Right		yawRotation
Left Look Ratchet		toggleLeftBumper
Right Look Ratchet		toggleRightBumper
Toggle Run Mode		toggleLeftThumbstickClick
Center Camera Forward		toggleRightThumbstickClick

Movement Type	Inputs	Related Default InputAction (Section 4.4)
---------------	--------	---

Oculus Touch

Movement Type	Inputs	Related Default InputAction (Section 4.4)
Move Forward	<ul style="list-style-type: none"> Left Thumbstick Y (up) 	zAxisMovement
Move Backward	<ul style="list-style-type: none"> Left Thumbstick Y (down) 	zAxisMovement
Move Left	<ul style="list-style-type: none"> Left Thumbstick X (left) 	xAxisMovement
Move Right	<ul style="list-style-type: none"> Left Thumbstick X (right) 	xAxisMovement
Look Up	<ul style="list-style-type: none"> Right Thumbstick Y (up) 	pitchRotation
Look Down	<ul style="list-style-type: none"> Right Thumbstick Y (down) 	pitchRotation
Look Left	<ul style="list-style-type: none"> Right Thumbstick X (left) 	yawRotation
Look Right	<ul style="list-style-type: none"> Right Thumbstick X (right) 	yawRotation
Left Look Ratchet		toggleLeftBumper
Right Look Ratchet		toggleRightBumper
Toggle Run Mode	<div>  Be Aware If Is Run Enabled (Section 3.3.3) is activated </div> <ul style="list-style-type: none"> Left Thumbstick (click) 	toggleLeftThumbstickClick
Center Camera Forward	<ul style="list-style-type: none"> Right Thumbstick (click) 	toggleRightThumbstickClick

4.4 Default Input Actions

The following table lists the Default Input Actions that are automatically created and initialized by the **VRSimulator**. These are fully customizable, should you so choose - for more information, please see **Configuring Input & Movement (Section 4)**.



You Should Know...

The InputActions described below are all defined and mapped to the input devices as described. However, the **VRSimulator** does not (currently) respond to all InputActions.

If an InputAction in the list below does not have any handler, then it will still fire a related event when detected - which if you want to handle, you can do so using a **custom input handler (Section 4.1)**.

For More Information...

- Custom Input Handling (Section 4.1)
- Order of Events (Section 5.3)
- Event Reference (Section 5.2)

InputAction Name	Movement	Fires Event (within EventManger ('EventManager Class' in the on-line documentation))	Inputs
togglePauseMenu		OnPauseButtonActivation ('OnPauseButtonActivation Event' in the on-line documentation) OnPauseButtonDeactivation ('OnPauseButtonDeactivation Event' in the on-line documentation)	<ul style="list-style-type: none"> • Keyboard: Escape • Mouse: • Gamepad: Start • HTC/Vive Wand: Menu Button • Oculus Remote: • Oculus Touch: Right Thumbstick Press
toggleView		OnViewButtonActivation ('OnViewButtonActivation Event' in the on-line documentation) OnViewButtonDeactivation ('OnViewButtonDeactivation Event' in the on-line documentation)	<ul style="list-style-type: none"> • Keyboard: Tab • Mouse: • Gamepad: Back / Menu • HTC/Vive Wand: • Oculus Remote: • Oculus Touch:
togglePrimaryTrigger		OnPrimaryTriggerActivation ('OnPrimaryTriggerActivation Event' in the on-line documentation) OnPrimaryTriggerDeactivation ('OnPrimaryTriggerDeactivation Event' in the on-line documentation)	<ul style="list-style-type: none"> • Keyboard: Left Control • Mouse: Left Button • Gamepad: Trigger based on Primary Gamepad Trigger (Section 3.3.3). • HTC/Vive Wand: Trigger on controller based on Primary Gamepad Trigger (Section 3.3.3). • Oculus Remote: Button.One (center D-

InputAction Name	Movement	Fires Event (within EventManger ('EventManager Class' in the on-line documentation))	Inputs
			Pad) <ul style="list-style-type: none"> ● Oculus Touch: Trigger based on Primary Gamepad Trigger (Section 3.3.3).
toggleSecondaryTrigger		OnSecondaryTriggerActivation ('OnSecondaryTriggerActivation Event' in the on-line documentation) OnSecondaryTriggerDeactivation ('OnSecondaryTriggerDeactivation Event' in the on-line documentation)	<ul style="list-style-type: none"> ● Keyboard: Left Alt ● Mouse: Right Button ● Gamepad: Opposite of Primary Gamepad Trigger (Section 3.3.3). ● HTC/Vive Wand: Opposite of Primary Gamepad Trigger (Section 3.3.3). ● Oculus Remote: ● Oculus Touch: Opposite of Primary Gamepad Trigger (Section 3.3.3)
toggleRightThumbstickClick	Rotates camera to point forwards.	OnRightThumbstickClickActivation ('OnRightThumbstickClickActivation Event' in the on-line documentation) OnRightThumbstickClickDeactivation ('OnRightThumbstickClickDeactivation Event' in the on-line documentation)	<ul style="list-style-type: none"> ● Keyboard: F, Numpad 5 ● Mouse: Center Button ● Gamepad: Right Thumbstick (click) ● HTC/Vive Wand: Right Grip ● Oculus Remote: ● Oculus Touch: Right Thumbstick (click)
toggleLeftThumbstickClick	Toggles Is Run Active on / off (see MovementManager (Section 3.3.3) settings).	OnLeftThumbstickClickActivation ('OnLeftThumbstickClickActivation Event' in the on-line documentation) OnLeftThumbstickClickDeactivation ('OnLeftThumbstickClickDeactivation Event' in the on-line documentation)	<ul style="list-style-type: none"> ● Keyboard: C, Left Shift ● Mouse: ● Gamepad: Left Thumbstick (click) ● HTC/Vive Wand: Left Grip ● Oculus Remote: ● Oculus Touch: Left Thumbstick (click)
toggleLeftBumper	Ratchets the camera to the left (rotates to the left by the number of degrees configured in Keyboard Rotation Ratchet (Section 3.3.3)).	OnLeftBumperActivation ('OnLeftBumperActivation Event' in the on-line documentation) OnLeftBumperDeactivation ('OnLeftBumperDeactivation Event' in the on-line documentation)	<ul style="list-style-type: none"> ● Keyboard: Q, Numpad 7 ● Mouse: ● Gamepad: Left Bumper ● HTC/Vive Wand: ● Oculus Remote: ● Oculus Touch:

InputAction Name	Movement	Fires Event (within EventManger ('EventManager Class' in the on-line documentation))	Inputs
toggleRightBumper	Ratchets the camera to the right (rotates to the right by the number of degrees configured in Keyboard Rotation Ratchet (Section 3.3.3)).	OnRightBumperActivation ('OnRightBumperActivation Event' in the on-line documentation) OnRightBumperDeactivation ('OnRightBumperDeactivation Event' in the on-line documentation)	<ul style="list-style-type: none"> • Keyboard: E, Numpad 9 • Mouse: • Gamepad: Right Bumper • HTC/Vive Wand: • Oculus Remote: • Oculus Touch:
toggleSelectionButton		OnSelectionButtonActivation ('OnSelectionButtonActivation Event' in the on-line documentation) OnSelectionButtonDeactivation ('OnSelectionButtonDeactivation Event' in the on-line documentation)	<ul style="list-style-type: none"> • Keyboard: Enter, Numpad Enter, Space • Mouse: • Gamepad: A • HTC/Vive Wand: Touchpad Center • Oculus Remote: • Oculus Touch: A
toggleCancelButton		OnCancelButtonActivation ('OnCancelButtonActivation Event' in the on-line documentation) OnCancelButtonDeactivation ('OnCancelButtonDeactivation Event' in the on-line documentation)	<ul style="list-style-type: none"> • Keyboard: X, Numpad Plus, Backspace • Gamepad: B • HTC/Vive Wand: • Oculus Remote: Back • Oculus Touch: B
toggleSecondaryButton		OnSecondaryButtonActivation ('OnSecondaryButtonActivation Event' in the on-line documentation) OnSecondaryButtonDeactivation ('OnSecondaryButtonDeactivation Event' in the on-line documentation)	<ul style="list-style-type: none"> • Keyboard: Page Up, Right Shift, Numpad 0, CapsLock • Mouse: • Gamepad: X • HTC/Vive Wand: • Oculus Remote: • Oculus Touch: X
toggleTertiaryButton		OnTertiaryButtonActivation ('OnTertiaryButtonActivation Event' in the on-line documentation) OnTertiaryButtonDeactivation ('OnTertiaryButtonDeactivation Event' in the on-line documentation)	<ul style="list-style-type: none"> • Keyboard: Numpad /, 2, K • Mouse: • Gamepad: Y • HTC/Vive Wand: • Oculus Remote: • Oculus Touch: Y
xAxisMovement	Used to control lateral movement (strafe) along the X-Axis.	OnXAxisMovementStart ('OnXAxisMovementStart Event' in the on-line documentation) OnXAxisMovementEnd ('OnXAxisMovementEnd Event' in the on-line documentation)	<ul style="list-style-type: none"> • Keyboard (if Is Strafe Enabled (Section 3.3.3)): A, D, Left Arrow, Right Arrow, Numpad 4, Numpad 6 • Mouse: • Gamepad: Left

InputAction Name	Movement	Fires Event (within EventManger ('EventManager Class' in the on-line documentation))	Inputs
			<p>Thumbstick X (left/right), D-Pad X (left/right)</p> <ul style="list-style-type: none"> • HTC/Vive Wand: • Oculus Remote: D-Pad Left, D-Pad Right • Oculus Touch: Left Thumbstick (left/right)
zAxisMovement	Used to control forward/backward movement along the Z-axis.	<p>OnZAxisMovementStart ('OnZAxisMovementStart Event' in the on-line documentation)</p> <p>OnZAxisMovementEnd ('OnZAxisMovementEnd Event' in the on-line documentation)</p>	<ul style="list-style-type: none"> • Keyboard: W, S, Up Arrow, Down Arrow, Numpad 8, Numpad 2 • Mouse: • Gamepad: Left Thumbstick Y (up/down), D-Pad Y (up/down) • HTC/Vive Wand: • Oculus Remote: D-Pad Up, D-Pad Down • Oculus Touch: Left Thumbstick Y(up/down)
pitchRotation	Used to look up or down (pitch).	<p>OnPitchRotationStart ('OnPitchRotationStart Event' in the on-line documentation)</p> <p>OnPitchRotationEnd ('OnPitchRotationEnd Event' in the on-line documentation)</p>	<ul style="list-style-type: none"> • Keyboard: • Mouse: Vertical • Gamepad: Right Thumbstick Y (up/down) • HTC/Vive Wand: • Oculus Remote: • Oculus Touch:
yawRotation	Used to look left or right (yaw).	<p>OnYawRotationStart ('OnYawRotationStart Event' in the on-line documentation)</p> <p>OnYawRotationEnd ('OnYawRotationEnd Event' in the on-line documentation)</p>	<ul style="list-style-type: none"> • Keyboard (if Is Strafe Enabled (Section 3.3.3) is deactivated): A, D, Left Arrow, Right Arrow, Numpad 4, Numpad 6 • Mouse: Horizontal • Gamepad: Right Thumbstick X (left/right) • HTC/Vive Wand: • Oculus Remote: • Oculus Touch:

5 Introduction to Scripting for the VRSimulator

The **VRSimulator** is designed to integrate fairly seamlessly into your VR experience. Remember, it's not meant to be a production-grade asset, however it is meant to be extensible and flexible to help you simulate your VR experience as comprehensively as possible.

Scripting Reference & Intellisense

We have thoroughly documented all of the public/static members, properties, methods, classes, etc. that are included in the **VRSimulator**. This scripting reference is available to you:

- **here in this user manual ('Immerseum.VRSimulator Namespace' in the on-line documentation)**, and;
- within Visual Studio Intellisense.

In Visual Studio, you can get helpful suggestions, guidance, and commentary on the VRSimulator's exposed properties and methods just by starting to type. Visual Studio will provide helpful tooltips and suggestions so that you can both autocomplete bits of code and peek into the underlying documentation about whatever you're working with.

Namespaces

The **VRSimulator** is designed to operate within the **Immerseum** namespace, and it too has received a namespace of its own (perhaps unsurprisingly, **Immerseum.VRSimulator**).

When scripting with the **VRSimulator**, you can either reference stuff using its fully formed name or you can also use the handy C# **using directive**:

Accessing Something from the VRSimulator

```
// Fully-formed approach - kind of long, difficult to read, and annoying.
using UnityEngine;

public class myClass : MonoBehaviour {
    bool hmdConnected = Immerseum.VRSimulator.HMDSimulator.isHMDConnected;
}

// With a using directive - short and sweet.
using Immerseum.VRSimulator {
    bool hmdConnected = HMDSimulator.isHMDConnected;
}
```

Design Approach

When designing the VRSimulator, we have adopted a number of design approaches that have a significant impact on our scripting interfaces:

1. **Singletons where we can.** Logically, you should only ever need one VRSimulator in your VR scene. Therefore, we're enforcing this in our design of the VRSimulator, where by design it applies the singleton pattern wherever appropriate.

2. **Static where we can.** In order to make the **VRSimulator** hard to break and easy to work with, our classes only expose the properties that you're likely to use. For those classes that apply the singleton pattern, those properties are exposed as static properties so you don't need to worry about navigating to instances and the like.
3. **Read-only properties are good.** Many of the properties exposed by the VRSimulator are the kind of properties that are calculated on-the-fly based on the state of your system, the state of your user, the state of the VRSimulator, etc. Since they're the kind of properties you'd never need to set, they're exposed only as read-only properties. **This includes the configuration settings.**
4. **Configuration (Section 3) via the editor.** Applying our philosophy of read-only properties are good properties, we don't let you edit (most) configuration settings programmatically. Configuration settings should be managed through the [Unity Inspector](#) only to maintain stability and maintainability of your VR scene. (See: [Configuring the VR Simulator \(Section 3\)](#))
5. **I love a good Event.** The VRSimulator is built on an event-driven model. Many of these events are publicly exposed, which allow you to "piggyback" on the VRSimulator's events and respond accordingly. This makes it really easy for you to layer your functionality in on top of the VRSimulator's, and thus make a really seamless integration between your code and our code. (See: [Event Reference \(Section 5.2\)](#))
6. **Abstracted and Extensible Input & Movement.** We've invested a fair bit of effort to come up with a comprehensive input and movement system that - by default - should be able to simulate most kinds of movement in your VR environment. But we know you might want to do your own thing and get creative with your inputs, with your reactions to inputs, with your movement, with everything. That's why we built an abstracted input / movement model that lets you modify, extend, or replace it altogether. (See: [Configuring Input & Movement \(Section 4\)](#))

5.1 API Scripting Reference

You Should Know...

Are you looking for a detailed API-level scripting reference? Well, this PDF user manual doesn't have one (it would be way too large to include here).

However, you're in luck! You can find a web-based version of our scripting reference at:

<https://developers.immerseum.io/vrsimulator>.

And if you really, really want to download a PDF of the scripting reference, you can do so there too.

5.2 Event Reference

The **VRSimulator** is internally structured around a comprehensive event system. Methods can listen or subscribe to events, and then when an event gets fired/invoked, the listener methods will execute. Events themselves can be invoked using static methods exposed by the **EventManager** (**'EventManager Class' in the on-line documentation**) singleton.



Please Note

A comprehensive diagram mapping the flow of events throughout the VRSimulator is in progress. When it is complete, you'll be able to find it on the **Order of Events (Section 5.3)** page.

Until it is ready, your best resource for understanding events is to review the events exposed by the **EventManager** (**'EventManager Class' in the on-line documentation**) in our scripting reference.

**For More
Information...**

- **Order of Events (Section 5.3)**
- **Scripting Reference: EventManager** (**'EventManager Class' in the on-line documentation**)

Listening for Events

Any method can "listen" for an event exposed by the VRSimulator **EventManager** (**'EventManager Class' in the on-line documentation**). A method that is listening for (subscribed to) an event will always execute whenever that event is fired/invoked. Below is a sample of a custom listener method that is subscribed to the **OnInputActionStart** (**'OnInputActionStart Event' in the on-line documentation**) event:

myCustomListener

```
public class myCustomClass : MonoBehaviour {
    void OnEnable() {
        // "Listen" for the Input Action.
        EventManager.OnInputActionStart += myCustomListener;
    }

    void OnDisable() {
        // "Unlisten" for the Input Action.
        EventManager.OnInputActionStart -= myCustomListener;
    }

    void myCustomListener(InputAction firedAction) {
        if (firedAction.name == "DesiredInputAction") {
            // Do something (or call another method) here.
        }
    }
}
```

In the code above, the `OnEnable()` method is where you set `myCustomListener` to "listen" for the event `EventManager.OnInputActionStart` ('`OnInputActionStart` Event' in the on-line documentation) to occur. Essentially, what you're doing is adding the method `myCustomListener` to a list of methods that will be called whenever `OnInputActionStart` occurs.

✓ Best Practice

To make our code easier to follow / keep straight, we always name our listener methods the same as the event they're listening to.

Thus, we wouldn't actually use `myCustomListener` but instead `OnInputActionStart`. Of course, you can use whatever naming conventions you like, but this one works well for us.

The `OnDisable()` method is where you then remove the `myCustomListener` method from the list of methods listening for `OnInputActionStart`. This is to prevent memory leaks - if you didn't "unlisten", then even when you closed Unity, shut down your game, and went off to get a cup of coffee there would still be a lonely little bit of software somewhere waiting in vain for the `OnInputActionStart` event to occur. That's a textbook definition of a nasty memory leak problem.

Reacting to an Event

The sample code above contains a method called `myCustomListener` which is a good example of reacting to an event. Here is that code again, just so we can go over its important parts:

myCustomListener

```
void myCustomListener(InputAction firedAction) {
    if (firedAction.name == "DesiredInputAction") {
        // Do something (or call another method) here.
    }
}
```

Please notice the return type and parameters for the method. **Every single** method that listens for the `OnInputActionStart` event **must** have the same return type (void) and the same parameters (a single `InputAction` ('`InputAction Class`' in the on-line documentation) object). However, that is not true for all events! Every single event expects its listeners to have a specific return type, and a specific set of parameters. To review those details, please review the event's documentation in the **Scripting Reference: EventManager** ('`EventManager Class`' in the on-line documentation).

! Be Careful!

If your custom listener for an event does not have the return type or parameters expected by the event, your code **will** throw an exception.

In this example, if the `OnInputActionStart` event is invoked, the `myCustomListener` method will be called. It will

check the `InputAction` passed to it against a particular string, and then execute a bunch of logic. This is just one example - what actually happens in an event listener method is entirely up to you!

Event Conventions

There are several design conventions that we have used when creating the VR Simulator's event system:

- **Naming Convention.** Events are named `On<Something>` where `<Something>` is a CamelCase description of what invoked the event. If the event has a start and an end, there will be two separate events: `On<Something>Start` and `On<Something>End`.
- **Default InputActions have associated events.** The **Default InputActions (Section 4.4)** supported by our input/movement system all have corresponding events associated with them. These events are fired whenever their related `InputAction` is detected. For events related either to toggles or button `InputActions`, the **MovementManager ('MovementManager Class' in the on-line documentation)** tracks the state of each `InputAction` and either calls `On<Input>Activation` or `On<Input>Deactivation` as appropriate.

5.3 Order of Events

You Should Know...

This page is a work in progress. Hopefully soon we'll have some pretty and easy-to-read diagrams.

In the meantime, you can find tons of information about the events generated by the VR Simulator by reviewing the scripting reference for the **EventManager ('EventManager Class' in the on-line documentation)** class.

Hopefully we'll have a nice diagram up soon!

**For More
Information...**

- **Scripting Reference: EventManager ('EventManager Class' in the on-line documentation)**

6 Release Notes

BETA-0.8.1

And our first bug fix and documentation fix!

- **IMPORTANT:** Our suppression of the reference errors does not work properly. As a result, you need to have both Oculus Utilities for Unity in your project, **and** the SteamVR plugin for the VRSimulator to work correctly.
- We've revised how certain inherited classes are organized in our script files. This is to deal with an intermittent and inconsistent Unity bug that apparently crops up now and again.

BETA-0.8

Our first release! This is the first public release of the **Immerseum SDK: VRSimulator**. Some important information follows:

Release and License

For the time being, the **VRSimulator** is released in BETA under a modified version of Unity's Asset Store End User License Agreement ([Unity's original version](#) | [our modified version](#)). Essentially, what our modified license says is that:

- You use the **VRSimulator** at your own risk. It is a beta-test product and we accept zero responsibility or liability for anything related to it (including but not limited to whether it works, how it affects your software, how it affects your computer, whether it will burn your house down, etc.).
- You're free to integrate the **VRSimulator** into any of your own code (i.e. your own games or VR experiences) and distribute that code once it's been compiled to whoever you want.
- What you're not allowed to do, however, is to provide the source code to the VRSimulator to any third parties as uncompiled source-code.
- You're also allowed to modify the VRSimulator's source code, provided that those modifications are not then distributed as source code.
- This license does not confer any rights to upgrades, support, or anything else that we may make available. If we decide to offer you something regarding the VRSimulator (like support or a new version for free), that is entirely our call.
- This license will always apply to this version of the software. Future versions may have wildly different licenses and be distributed under wildly different terms - but those new versions/licenses will have no impact on version: BETA-0.8.1.



Please Note

In the near future, we do plan to release the **VRSimulator** on the Unity Asset Store, which will put the **VRSimulator** under the terms of Unity's original [Asset Store EULA](#).

We are currently planning to charge for the **VRSimulator** when we make it available in the Unity Asset Store.

Requirements & Supported Devices

The **VR Simulator**'s requirements are very simple:

- Windows or MacOS

You Should Know...

While the **VRSimulator** was specifically specifically built to support Windows and MacOS, it has only been tested under Windows.

- Unity 5.3 or higher,
- Both the [SteamVR plugin](#) or the [Oculus Utilities for Unity](#) package in your Unity project, and;
- One or both of the following camera rigs in your scene:
 - The **SteamVR**: [CameraRig] prefab, and/or;
 - The **Oculus**: OVRCameraRig prefab (or the OVRPlayerController prefab - which also contains the OVRCameraRig prefab)

You Should Know...

By default, you can find the camera rig prefabs in the following locations:

SteamVR: Assets / SteamVR / Prefabs /

Oculus: Assets / OVR / Prefabs /

And that's it!

Documentation & Support

You can find extensive documentation and support for the **VRSimulator** on the Immerseum web site at:

<http://developers.immerseum.io/vr-simulator>

If you need help using the VRSimulator, we're here to help you! Reach out using any of the following:

- **Email** us at: support@immerseum.io
- Join our **Slack group**: immerseum-vr-dev.slack.com (you can get an invite by clicking here)
- **Tweet** at us: [@ImmerseumVR](https://twitter.com/ImmerseumVR)

Be Aware

We're a teensy, tiny team right now. The **VR Simulator** is a one-person project, so response times **will** vary. We'll do our best to get back to you ASAP, however.

7 Index

API Scripting Reference, 73
Configuring Input & Movement, 41-45
Configuring the VRSimulator, 26
ControllerManager Settings, 37-39
Copyright Page, 1
Custom Input Handling, 46-51
Custom InputAction Mapping, 52-60
Default Input Actions, 67-70
Default Movement Mapping, 61-66
Event Reference, 74-76
FAQ, 24-25
Getting Started with the VRSimulator, 7-17
HMDSimulator Settings, 36-37
InputActionManager Settings, 40
Introduction, 3-6
Introduction to Scripting for the VRSimulator, 71-72
Managing Your Camera, 27-29
Managing Your Controllers, 30-35
MovementManager Settings, 39-40
Order of Events, 77
Quick Start, 18-21
QuickStart, 18-21
Release Notes, 78-79
Requirements and Supported Devices, 22-23
Welcome, 3-6
Welcome to the VR Simulator, 3-6