OpenXR Actions And Spaces APIs In O3DE

Background, History and Developer's Tutorial

Galib F. Arrieta (galibzon@github, lumbermixalot@github, galibzon@discord)

FYI

This presentation is about the implementation of two aspects of the OpenXR spec in O3DE:

1. Spaces - https://registry.khronos.org/OpenXR/specs/1.0/html/xrspec.html#spaces
2. Input and Haptics - https://registry.khronos.org/OpenXR/specs/1.0/html/xrspec.html#input

OpenXR Spaces - Background

In simple terms, an OpenXR Space is just a Frame Of Reference (aka **Anchor**) encoded as a tuple: Name(string) + Transform (AZ::Transform).

There are two kinds of spaces in OpenXR: **Reference** Spaces and **Action** Spaces. Within the OpenXR API, all spaces are identified with the same opaque handle type, **XrSpace**. For O3DE developers, an OpenXR Space is identified by its **name (string)**.

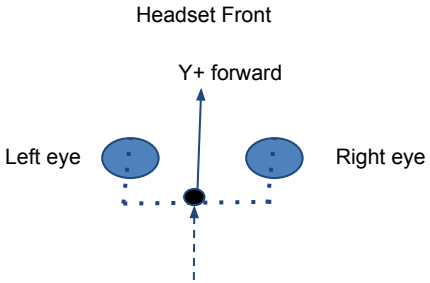
Reference Spaces are used by applications to bootstrap their **spatial reasoning,** while **Action** Spaces are used when reading user input related with Hands or Grip orientation.

OpenXR Well-Known Reference Spaces (1)

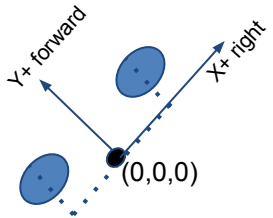
OpenXR defines a set of well-known **reference spaces** that applications use to bootstrap their **spatial reasoning**. Each reference space has a well-defined meaning, which establishes where its origin is positioned and how its axes are oriented.

1. **VIEW:** Defines the current pose of the Head Centroid (for Stereo systems). For XR applications running on a phone this could be the phone location + orientation.
2. **LOCAL:** Typically centered around the initial location and orientation upon system reset.
3. **STAGE.** Typically represents the real world **user-defined** Squared Boundary that defines the safe-to-play area.

OpenXR Well-Known Reference Spaces (2)

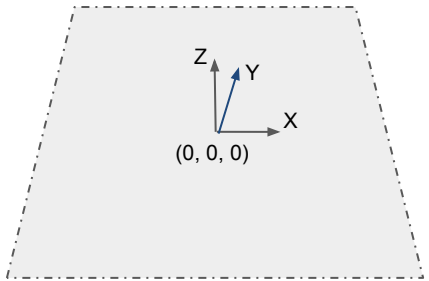


VIEW



Whatever was the orientation of the device upon reset becomes the (0,0,0) of the LOCAL reference space

LOCAL

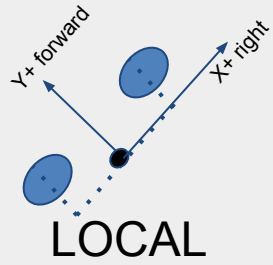


STAGE

REMARK: In O3DE, The OpenXR Space API conforms to the O3DE Axis Convention: X+ is right, Z+ is Up, Y+ is Forward.

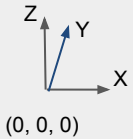
You can only query the location of a Reference Space with respect to **another** Reference Space.

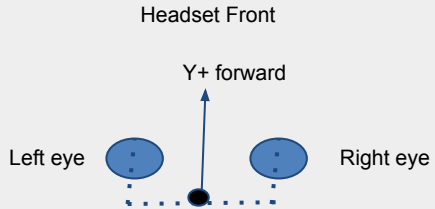
You can only query the location of a Reference Space with respect to **another** Reference Space.



LOCAL

The user was here upon Session reset





VIEW

```
local outcome = OpenXRReferenceSpaces.GetReferenceSpacePose(“View”, "Local")
```

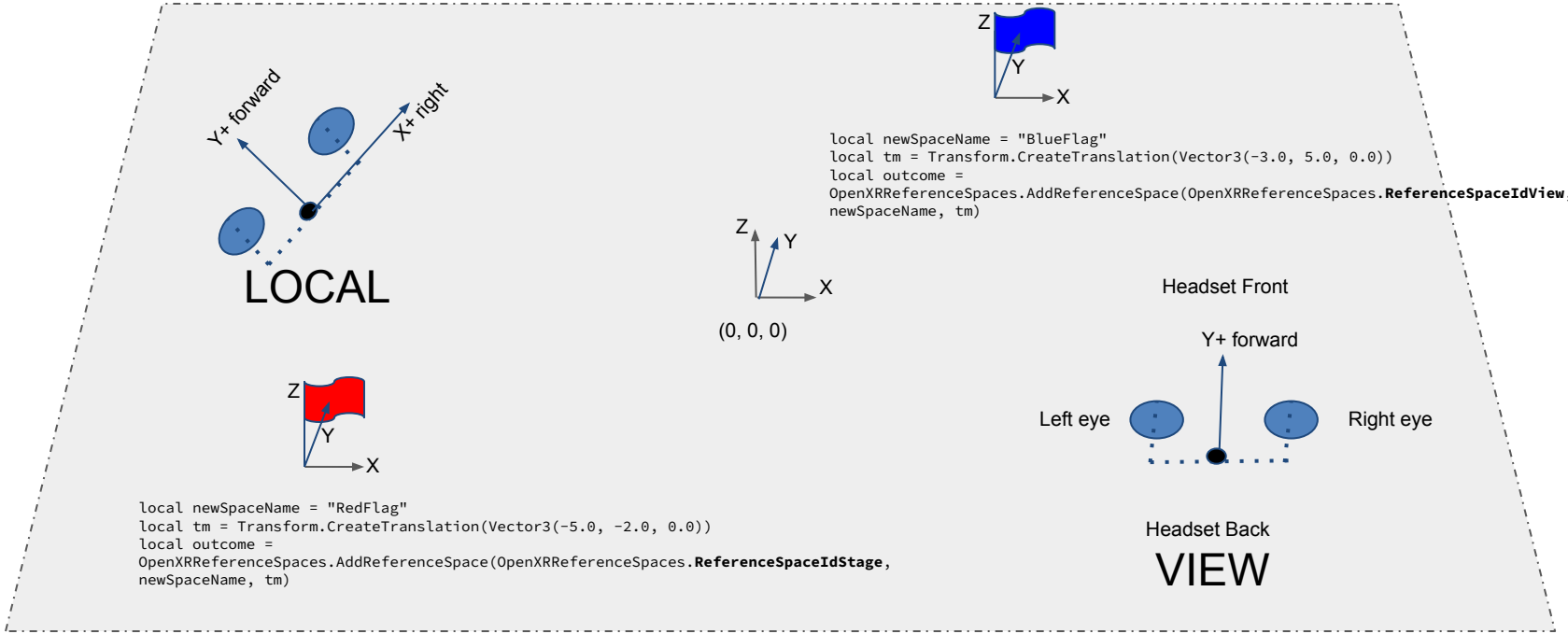
Returns a Transform that describes where is the VIEW with respect to the LOCAL space.

```
local outcome = OpenXRReferenceSpaces.GetReferenceSpacePose(“View”, "Stage")
```

Returns a Transform that describes where is the VIEW with respect to the STAGE space.

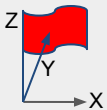
STAGE

You can create your own Reference Spaces (aka Anchors).



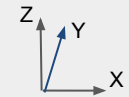
You can only query the location of a Reference Space with respect to **another** Reference Space.

LOCAL

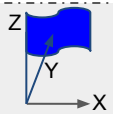



```
local outcome = OpenXRReferenceSpaces.GetReferenceSpacePose(“RedFlag”, "View")
```

Returns a Transform that describes where is the **“RedFlag”** with respect to the **“View”** space.



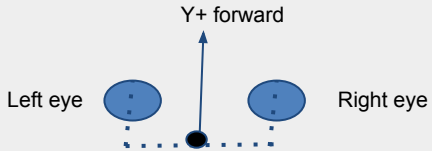
(0, 0, 0)




```
local outcome = OpenXRReferenceSpaces.GetReferenceSpacePose(“View”, "BlueFlag")
```

Returns a Transform that describes where is the **“View”** with respect to the **“BlueFlag”** space.

Headset Front



Headset Back

VIEW

STAGE

Where is the O3DE API for OpenXR Spaces?

OpenXRVk::OpenXRReferenceSpacesInterface (C++)

The C++ API is exposed in the BehaviorContext (Lua, ScriptCanvas, etc) in:
OpenXRVkBehaviorReflection.cpp (“OpenXRReferenceSpaces”)

An example Lua script is available at:
xr_spaces_api_test.lua

Inputs And Haptics - Background

OpenXR presents a new paradigm into how applications would manage user I/O.

For the last 20+ years hardware manufacturers and software developers have been building interactive applications according to the USB/Bluetooth HID, Human Interface Device, standard. The HID standard, and some API abstractions like Linux EVDEV, defines a set of devices like Gamepad, Keyboard, Mouse, etc and their expected buttons, trackpads, etc.

Bottom line, everybody had agreed to a well defined contract.

Inputs And Haptics - Background continued…

This well defined I/O contract, has a few limitations:

1. It was not possible for hardware manufacturers to design new hardware interfaces that provided complex user input data like Vector or Quaternion data. E.g. Location and orientation of the user hands, etc.
2. The standard is basically “written in stone”, because the button codes are shipped within **OS drivers** and **low level APIs like EVDEV**. Which means modifying the standard would need to be propagated around all HID library headers, etc.

The OpenXR Way

Regarding user I/O the key tenets for OpenXR are flexibility and extensibility. The API for querying the available hardware is now in user space, and it works in terms of **strings.** These strings look like UNIX directory paths.

Each equipment vendor advertise their “paths”, and applications would simply declare what “paths” are supported.

This approach opens the possibility of defining user I/O support in a **data driven** way, unlike hard-coded scan codes in OS drivers.

Let’s walk through an example

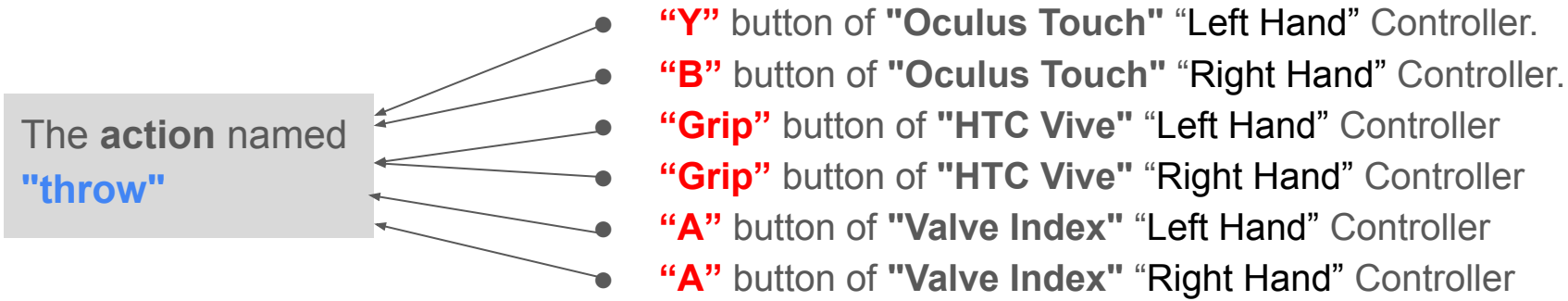
- Let's suppose you are developing an application where the user will be able to **throw** objects.
- Furthermore, let's assume that this action can be represented with a **boolean state**.

```
bool shouldThrow = GetBooleanActionState("throw");

if (shouldThrow) {

    Throw();

}
```

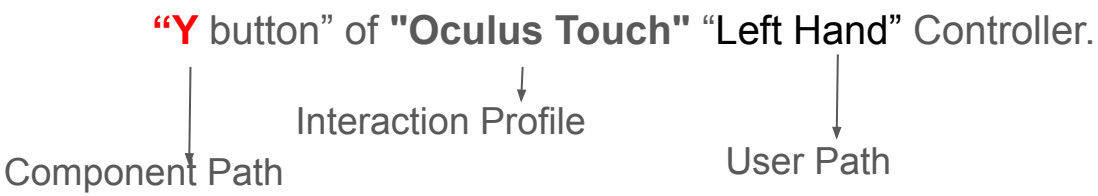
Example continued…

The first question you may be asking right now is how to map **"throw"** to a hardware button?

In OpenXR, you define your application I/O requirements as list of actions, where each action has **1-to-many relationship** with a set of hardware buttons identifiable by strings:



Breaking down the parts of a hardware button description in OpenXR



Interaction Profile:
Name: “Oculus Touch”
Path: "/interaction_profiles/oculus/touch_controller"

User Path:
Name: “Left Hand”
Path: "/user/hand/left"

Component Path:
Name: “Y button”
Path: "/input/y/click"

The **Name** is not important, and each application/engine can choose an arbitrary name to present to the user.

What matters is the **Path.** Each hardware vendor declares and publishes their paths for all the of their Interaction Profiles. These paths will become the unique identifiers for the low level OpenXR library.

https://registry.khronos.org/OpenXR/specs/1.0/html/xrspec.html#semantic-path-interaction-profiles

The Interaction Profile

From the point of view of XR/VR hardware vendors, an **Interaction Profile** is an abstraction that represents a family of headsets and controllers with common I/O functionality. For example, for **Meta™,** the **Oculus Touch Controller** profile, uniquely identifiable by its path: */interaction_profiles/oculus/touch_controller,* supports the following devices:

- Quest 1
- Quest 2
- Quest Pro
- Quest 3

https://registry.khronos.org/OpenXR/specs/1.0/html/xrspec.html#_oculus_touch_controller_profile

The User Path

Typically the User Path represents the controllers, joysticks or user hands supported under a particular **Interaction Profile**. The two most common User Paths supported by most Interaction Profiles are:

1. /user/hand/left
2. /user/hand/right

Not always it represents a hand or arm, for example the Microsoft Xbox Controller Profile, only supports one User Path, known as “/user/gamepad”.

The Component Path

To put it simply, a **Component Path** represents a **button** available in a particular **User Path** (aka Controller, Joystick or Hand).

Some **Component Paths** are **common** to all **User Paths** supported by an **Interaction Profile**. For example, the Component Path "*/input/trigger/value"* is supported by all User Paths (*"/user/hand/left"* and *"/user/hand/right"*) of the Oculus Touch Controller profile.

Some **Component Paths** are **unique** to an **User Path** supported by an **Interaction Profile.** For example, the Component Path "*/input/x/click"* is supported **only** by the User Path *"/user/hand/left"* of the Oculus Touch Controller profile.

Interaction Profiles In O3DE

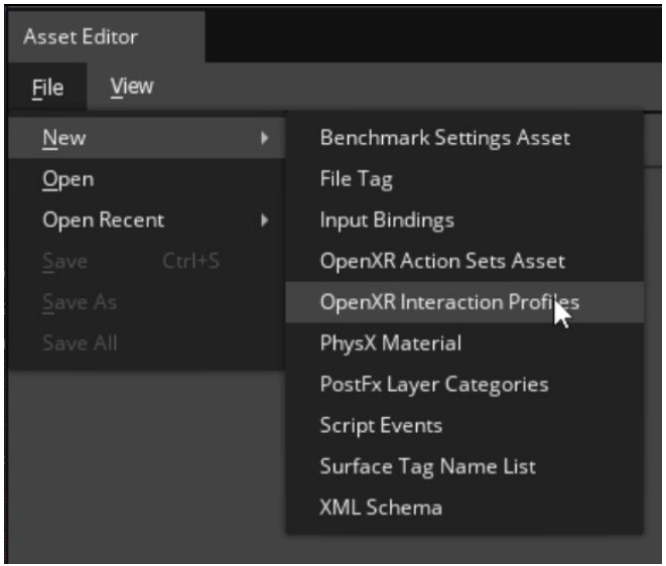
An asset file with extension “***.xrprofiles”**, contains **All** Interaction Profiles supported by any given O3DE-based Application. As new hardware is introduced in the future, the O3DE developer only needs to add more data to the same asset file. **No need to recompile the engine**.

The OpenXRVk Gem ships with an **Interaction Profiles Asset** under the relative path: “Assets/OpenXRVk/system.xrprofiles”.

Most developers can leverage this file or make a copy of it into their own projects. The key takeaway is that **this file changes very rarely**. It should only change when new Interaction Profiles are introduced into the market.

Editing An Interaction Profiles Asset In O3DE

To create or edit an Interaction Profiles asset, the OpenXRVk Gem conveniently exposes this asset type under the **Asset Editor**. The **Asset Editor** panel is available in the **Editor** from the menu: *Tools -> Asset Editor*

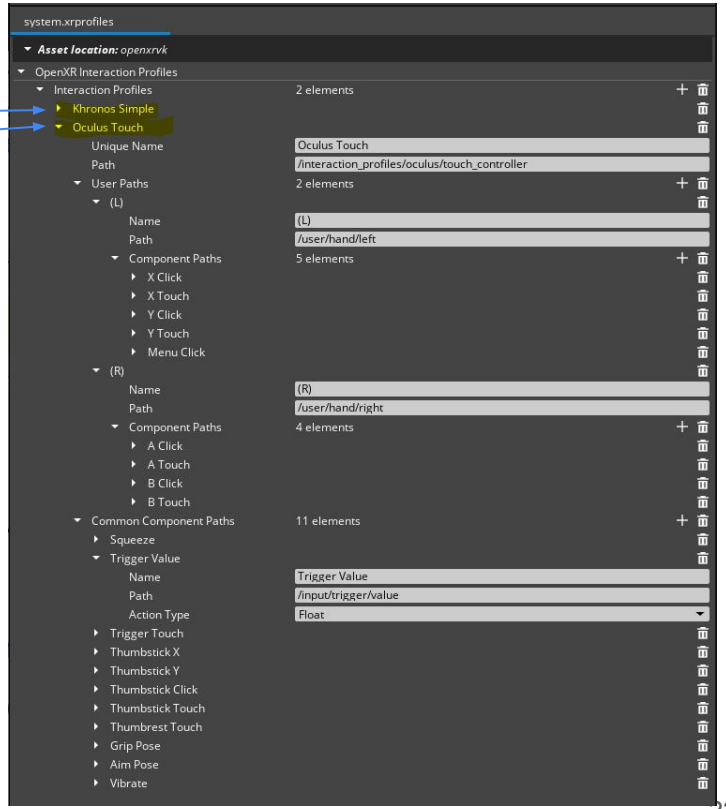


Breaking Down **system.xrprofiles** [OpenXRVk Gem] (1)

So far, contains two Interaction Profiles:

- Kronos Simple
- Oculus Touch

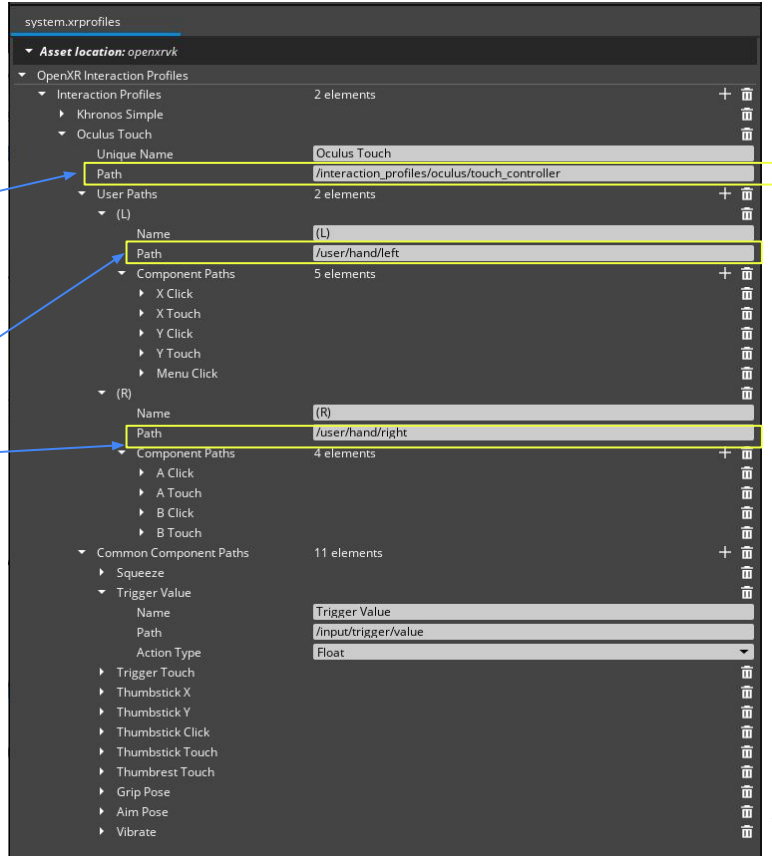
Feel free to add or more!



Breaking Down **system.xrprofiles** [OpenXRVk Gem] (2)

The name **“Oculus Touch”** is arbitrary, on the other hand, its path conforms to the OpenXR Spec:
https://registry.khronos.org/OpenXR/specs/1.0/html/xrspec.html#_oculus_touch_controller_profile

Per the spec, it contains only two User Paths. In this asset, their names are arbitrary “(L)” and “(R)”, but their **paths** come from the spec.



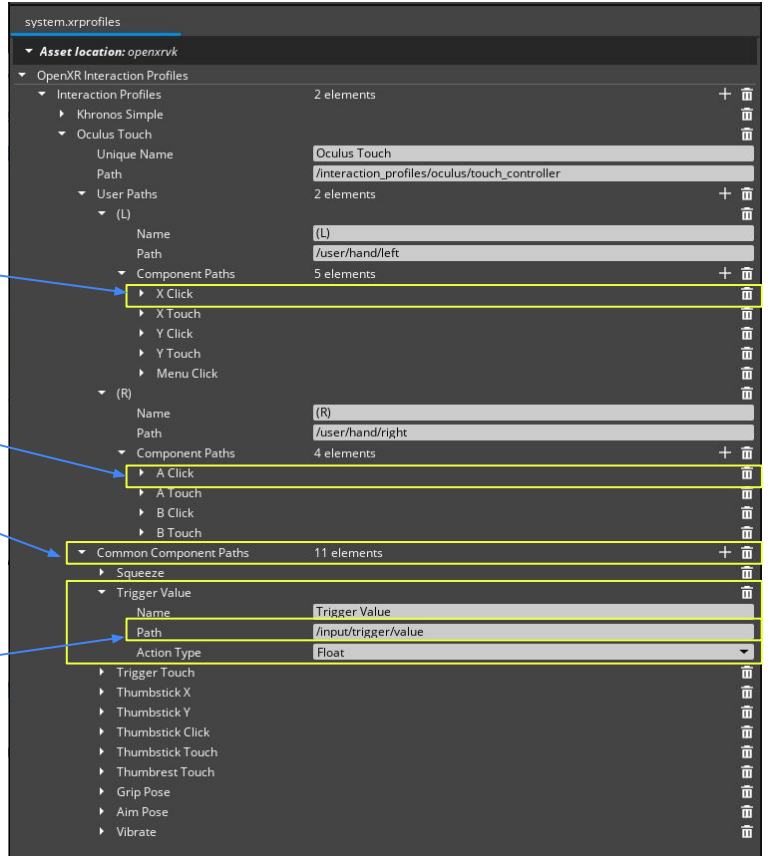
Breaking Down **system.xrprofiles** [OpenXRVk Gem] (3)

Some Component Paths (aka buttons) are **unique** to a particular User Path. Examples:

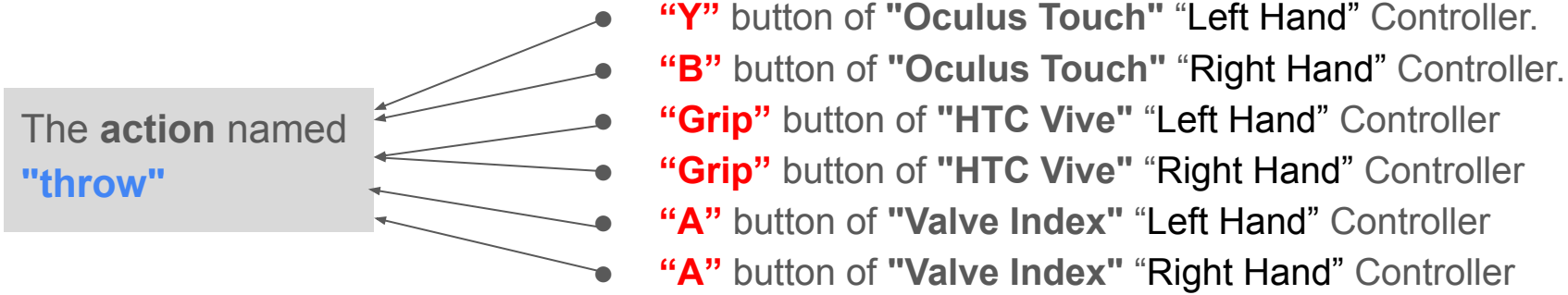
- The “X Click” button is unique to the “(L)” User Path.
- The “A Click” button is unique to the “(R)” User Path.

Some Component Paths are **common** to **both** User Paths.

As mentioned already, within this asset, all of the names are arbitrary, but the paths must conform to the OpenXR spec:
https://registry.khronos.org/OpenXR/specs/1.0/html/xrspec.html#_oculus_touch_controller_profile



Do you remember our starting example?



So far, we covered the **right side of the picture**… The list of available buttons that the application will be compatible with. We learned that the data listed here comes from an Interaction Profiles asset. E.g.system.xrprofiles.

Now we'll talk about the **left side of the picture…** The **bespoke** actions that the application will use to define the gameplay.

OpenXR Action Sets and Actions in O3DE.

In OpenXR not only you work with custom **Actions**, but you must group your actions into **Action Sets**.

Action Sets are **immutable** but they can be activated and deactivated at runtime. They also have a **priority** that will serve as tie breaker when several actions are mapped to the same buttons.

Here is an example of three Action Sets

Action Set
Name: “ui”
Priority: 1
Actions:

- “select”
- “move_cursor”
- “cancel”

Action Set
Name: “gameplay”
Priority: 2
Actions:

- “throw”
- “jump”
- “grab”
- “kick”

Action Set
Name: “game_state”
Priority: 2
Actions:

- “pause_resume”
- “quit”

The Action Sets Asset

In O3DE, you define all Action Sets (and their Actions) in an asset with extension ***.xractions.**

By default, the OpenXRVk Gem will load the Action Sets asset located at: "OpenXRVk/default.xractions".

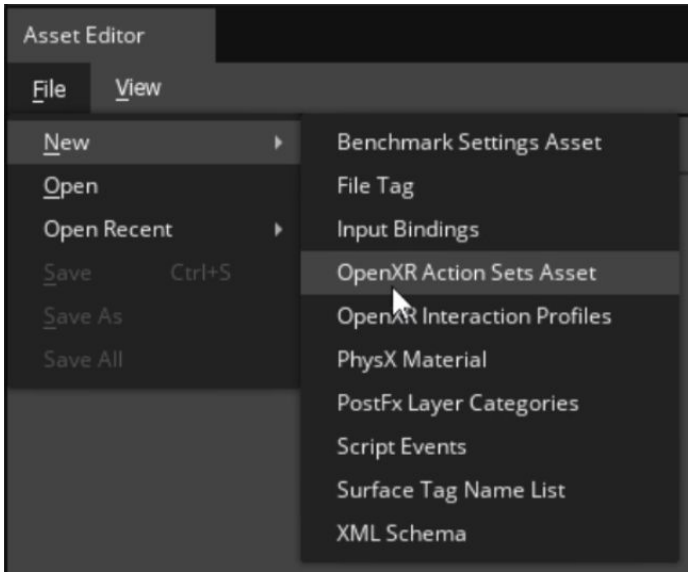
Applications can override which Action Sets asset to load with the registry key: "*/O3DE/Atom/OpenXR/ActionSetsAsset*".

Example for a project called **AdventureVR. "***AdventureVR/Registry/adventurevr.setreg":*

```
{
    "O3DE": {
        "Atom": {
            "OpenXR": {
                "ActionSetsAsset": "assets/adventurevr.xractions"
            }
        }
    }
}
```

Editing An Action Sets Asset In O3DE

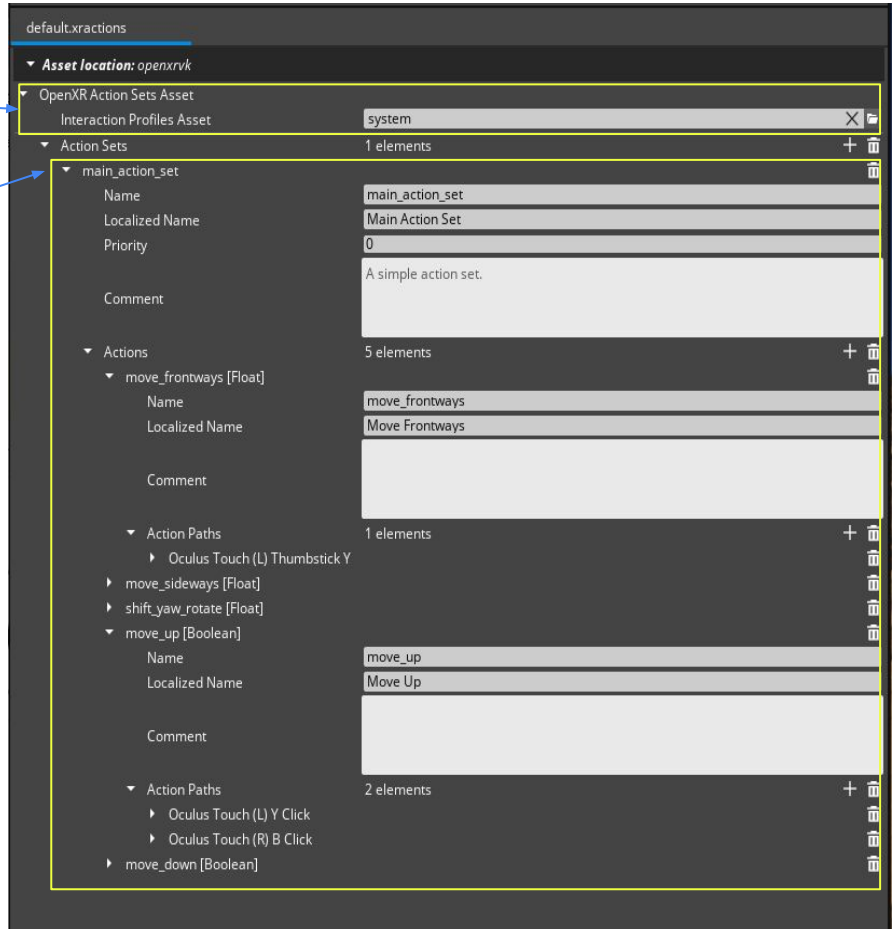
To create or edit an Action Sets asset, the OpenXRVk Gem conveniently exposes this asset type under the **Asset Editor**. The **Asset Editor** panel is available in the **Editor** from the menu: *Tools -> Asset Editor*



Breaking Down **default.xractions** [OpenXRVk Gem] (1)

Must reference an Interaction Profiles asset.
In this case it uses **system.xrprofiles**

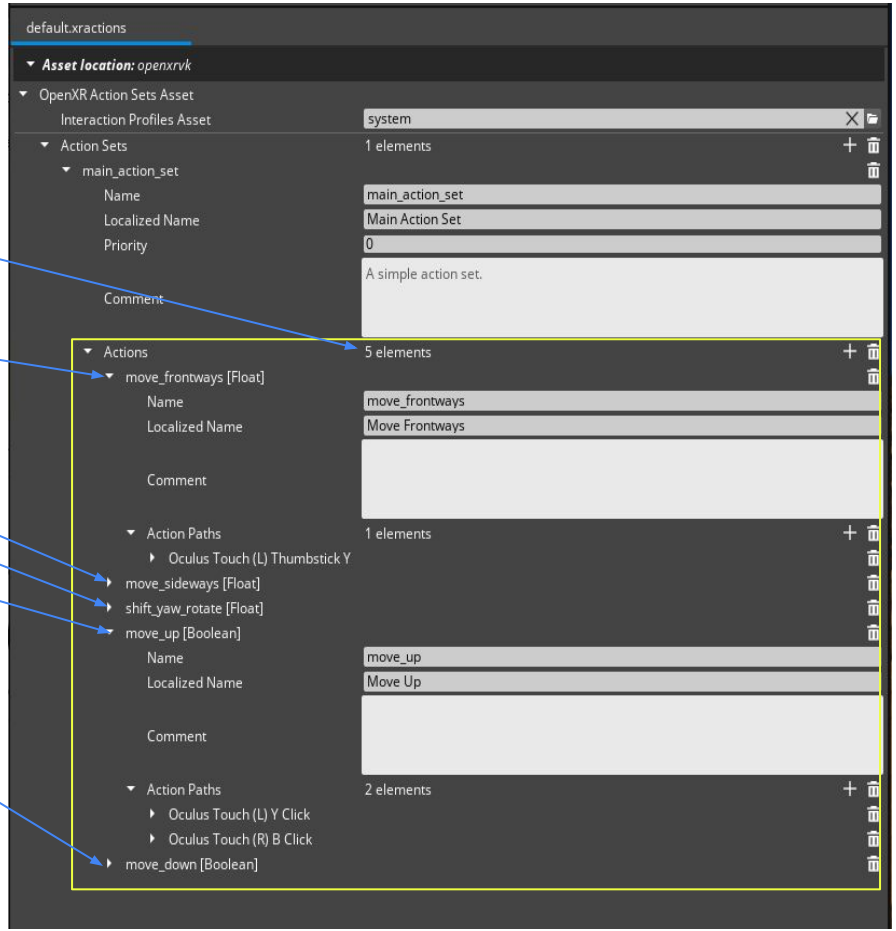
So far, only one Action Set is defined in this asset. Its name is “**main_action_set**”



Breaking Down **default.xractions** [OpenXRVk Gem] (2)

"**main_action_set**" contains 5 actions:

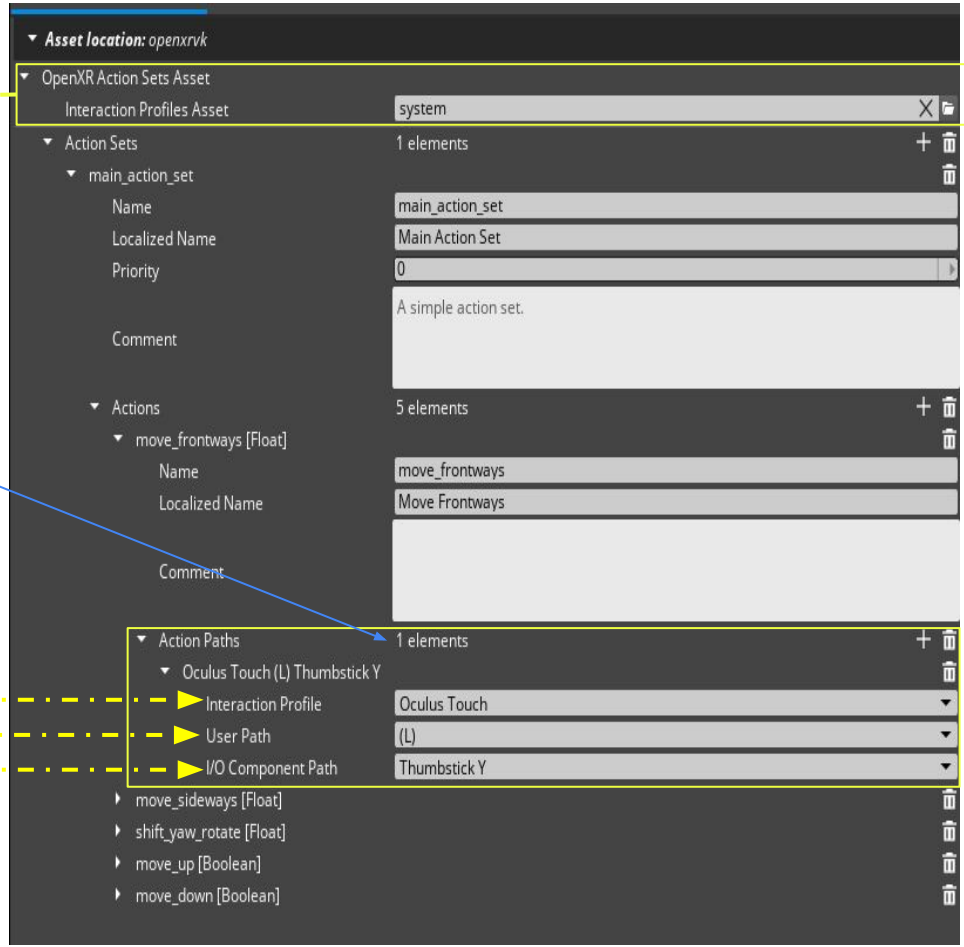
1. "move_frontways"
2. "move_sideways"
3. "shift_yaw_rotate"
4. "move_up"
5. "move_down"



Breaking Down **default.xractions** [OpenXRVk Gem] (3)

"**move_frontways**" contains only 1 action path.

When defining the Action Path, the Combo Boxes are populated with data from the **referenced** Interaction Profiles asset.



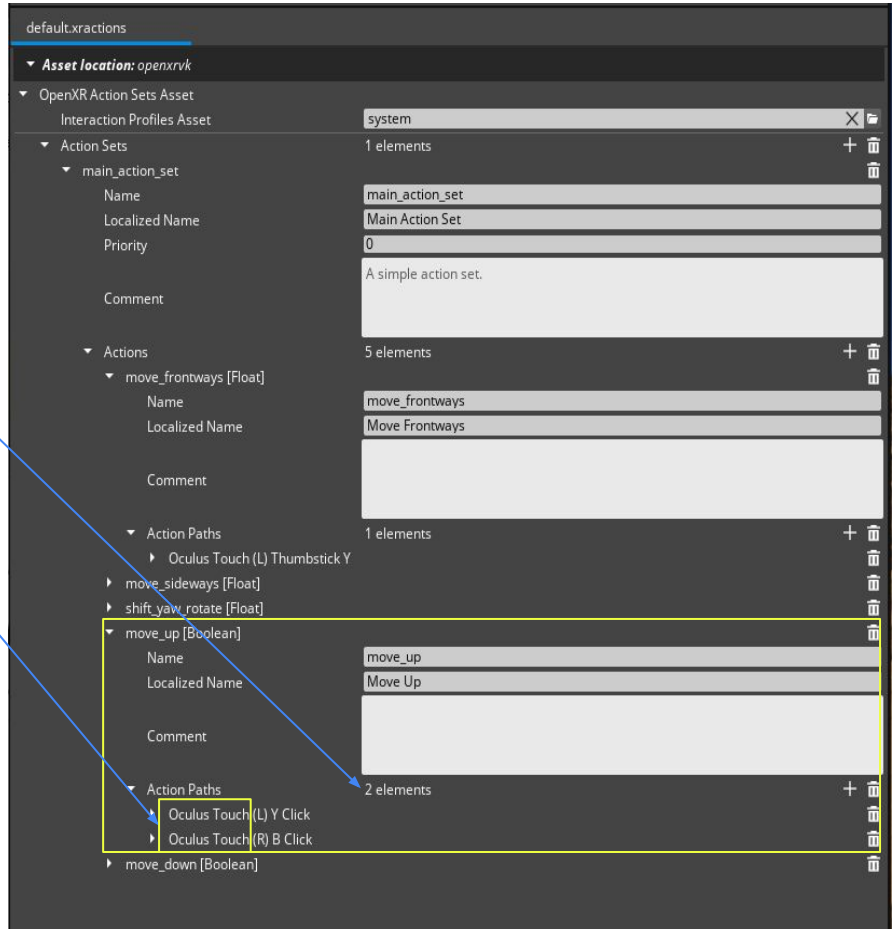
Breaking Down **default.xractions** [OpenXRVk Gem] (4)

“**move_up**” contains 2 action paths.

Both action paths refer to different buttons under the **same Interaction Profile**.

REMARK 1: Other buttons under other Interaction Profiles could have been added, which would make the application compatible with other VR equipment.

REMARK 2: We can only add action paths with data that comes from the referenced **system.xrprofiles.**



Runtime Example With Lua (1)

O3DE Applications Will Use the following API to work with OpenXR Actions: OpenXRVkActionsInterface.h

```
using OpenXRActionsInterface = AZ::Interface<IOpenXRActions>;
```

OpenXRActionsInterface is exposed into the Behavior Context (For Lua, ScriptCanvas, etc) in this file: *OpenXRVkBehaviorReflection.cpp ("OpenXRActions")*

An example Lua script is provided at this location: xr_camera_move.lua

REMARK: This Lua script can be used as a replacement of the C++ component called XRCameraMovementComponent.cpp

Runtime Example With Lua (2)

```
function xr_camera_move:OnActivate()
  -- OPTIONAL: Cache all action handles
  self._moveFrontwaysHandle = OpenXRActions.GetActionHandle("main_action_set", "move_frontways")
  self._moveUpHandle = OpenXRActions.GetActionHandle("main_action_set", "move_up")
end
```

```
function xr_camera_move:OnTick(deltaTime, timePoint)
  self._cameraMovementStates = Vector3(0.0, 0.0, 0.0)

  local outcome = OpenXRActions.GetActionStateFloat(self._moveFrontwaysHandle)
  if outcome:IsSuccess() then-- Typically, a failed outcome means the joystick is resting.
    self._cameraMovementStates.y =  outcome:GetValue()
  end

  outcome = OpenXRActions.GetActionStateBoolean(self._moveUpHandle)
  if outcome:IsSuccess() then-- Typically, a failed outcome means the joystick is resting.
    if outcome:GetValue() then
      self._cameraMovementStates.z = 1.0
    end
  end
end
```

Summary: OpenXR Actions In O3DE

1. **Very rarely**: Extend **system.xrprofiles**, or create your own using the **Asset Editor.** Only required when adding new hardware support.

2. **Once per application**: Define your own *.xractions file and specify it under the Registry Key: "`/O3DE/Atom/OpenXR/ActionSetsAsset`".

3. **At runtime**, for C++, use `OpenXRActionsInterface.` For Lua, use the API exposed in *OpenXRVkBehaviorReflection.cpp.*

OpenXR Actions And Spaces APIs In O3DE

The End

Questions?

Galib F. Arrieta (galibzon@github, lumbermixalot@github, galibzon@discord)