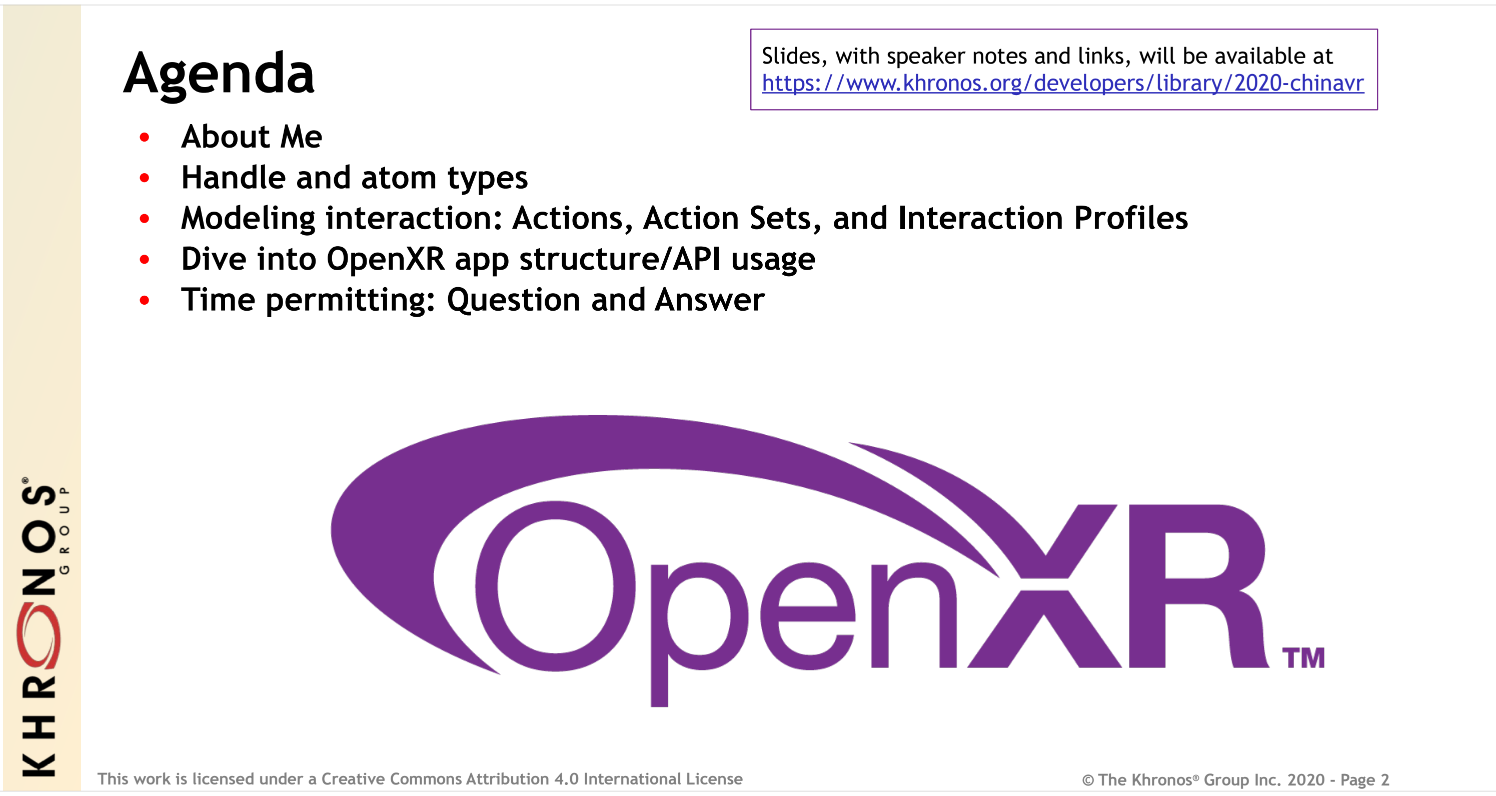
OpenXR Master Class

Ryan A. Pavlik, Ph.D.
Principal Software Engineer, Collabora, Ltd.
OpenXR Working Group Spec. Editor
ChinaVR, September 2020

Hello everyone, and welcome to our OpenXR master class.

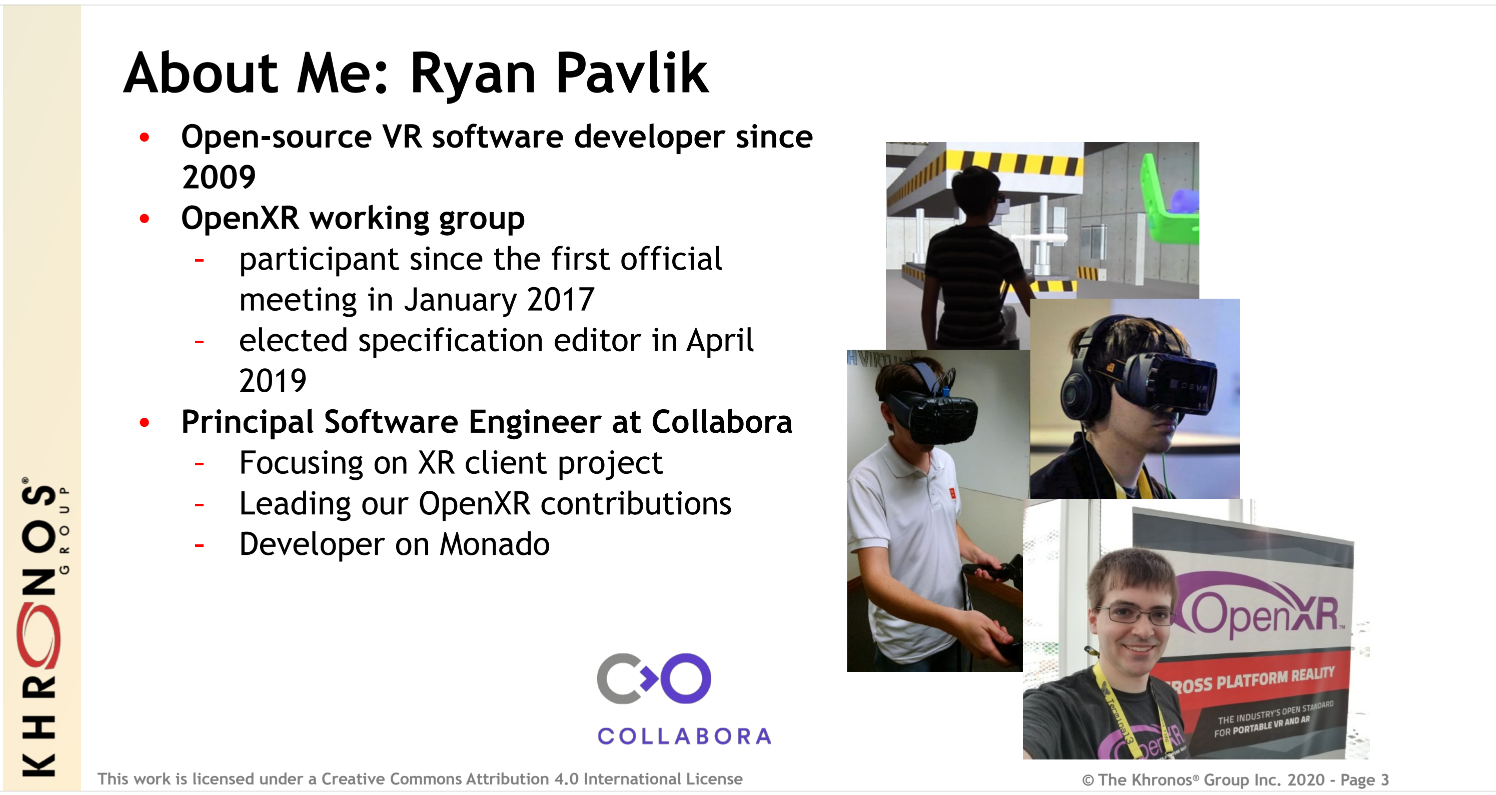


Agenda

Slides, with speaker notes and links, will be available at https://www.khronos.org/developers/library/2020-chinavr

- About Me
- Handle and atom types
- Modeling interaction: Actions, Action Sets, and Interaction Profiles
- Dive into OpenXR app structure/API usage
- Time permitting: Question and Answer

The plan for the talk today is to spend a few moments introducing myself, and discussing the basic object handle types in OpenXR. At that point, we'll have the background to explore how OpenXR models interaction. Then, we'll take a deep dive into OpenXR application structure and API usage. We should have time for questions and answers at the end.

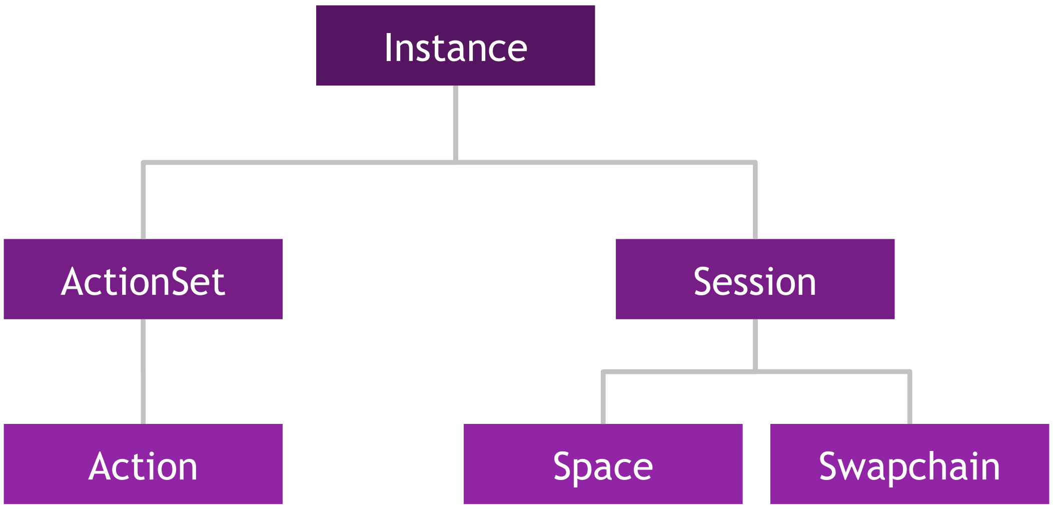


About Me: Ryan Pavlik

- **Open-source VR software developer since 2009**
- **OpenXR working group**
  - participant since the first official meeting in January 2017
  - elected specification editor in April 2019
- **Principal Software Engineer at Collabora**
  - Focusing on XR client project
  - Leading our OpenXR contributions
  - Developer on Monado

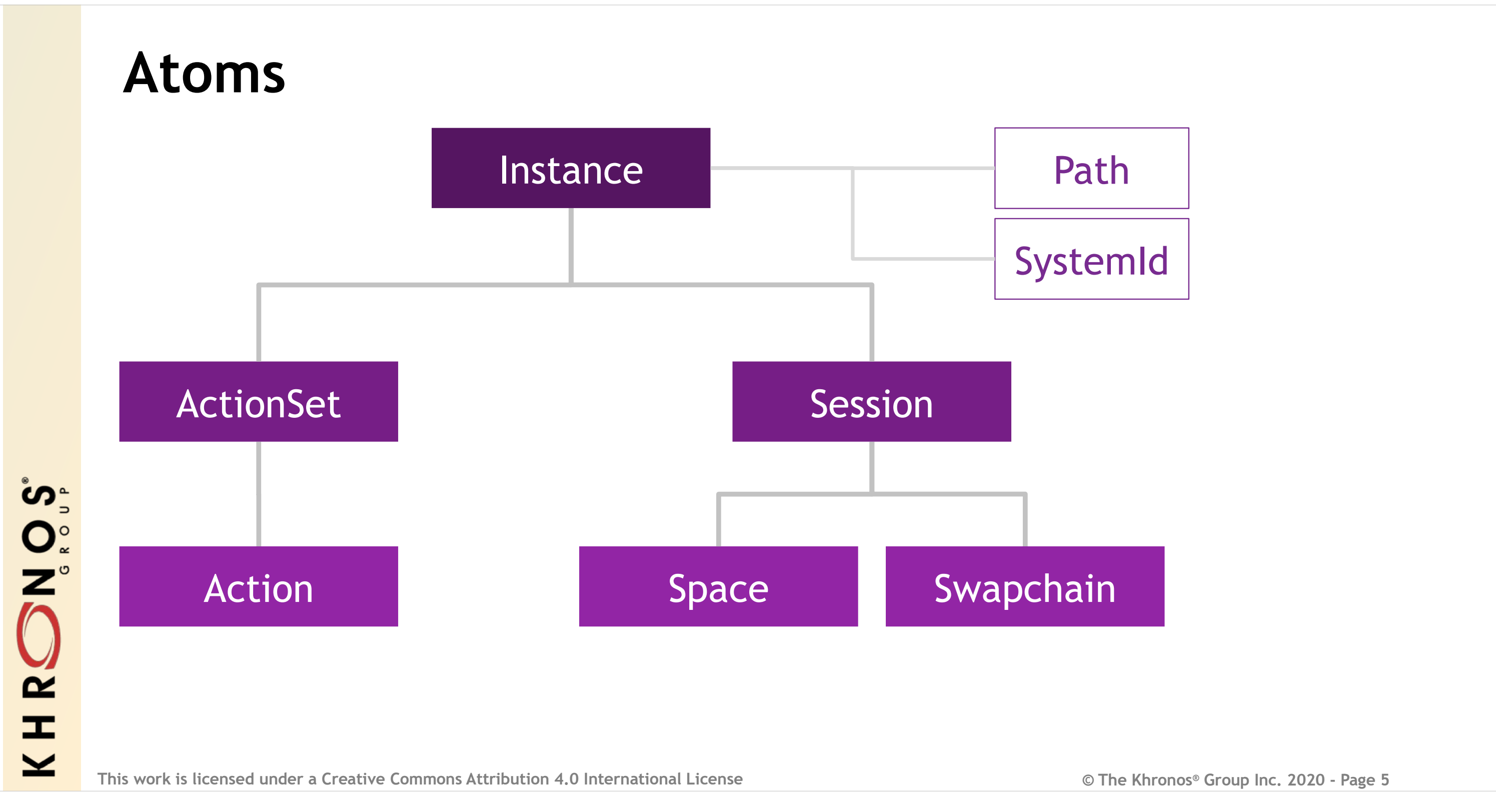
My name is Ryan Pavlik. I've been working in the VR realm since around 2009. I've been involved with the OpenXR working group since the first official meeting in 2017. I was elected specification editor for the OpenXR working group in 2019. I am a principal software engineer at Collabora, where I work on customer projects in the XR team, and contribute to Monado which is our OpenXR runtime.

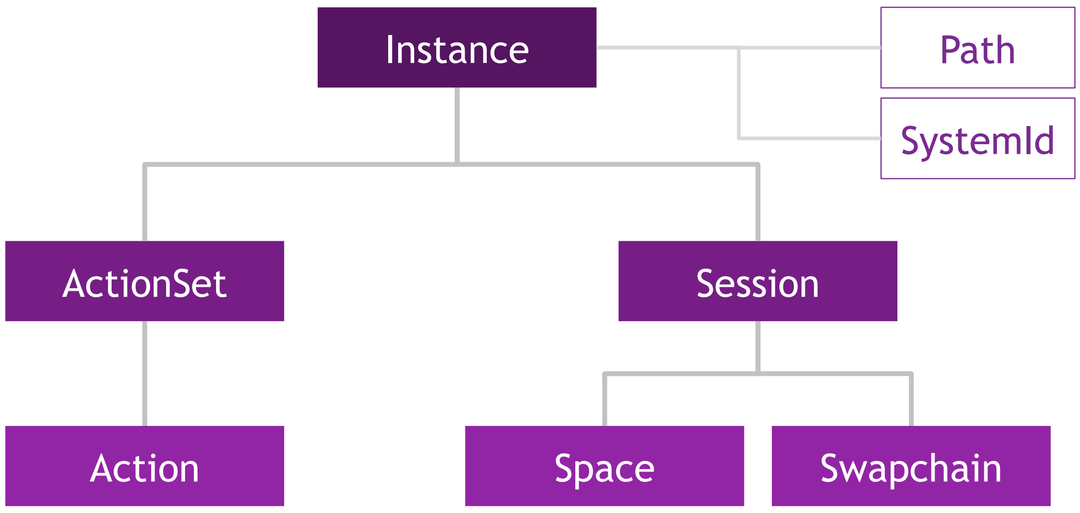
OpenXR Handle Types



There are a number of important objects in OpenXR, which are represented by handle types.

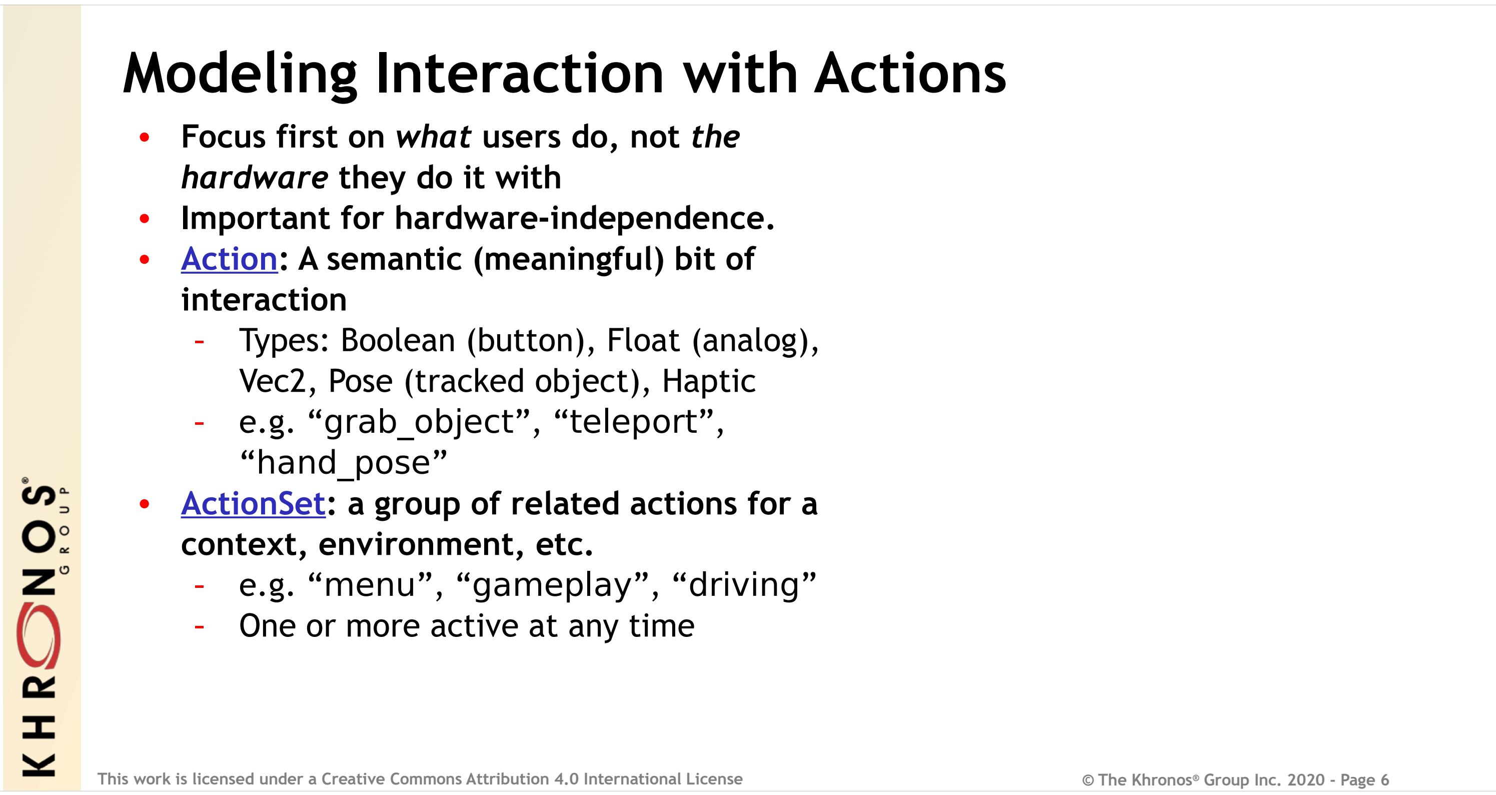
These are the main handle types. All handle types except XrInstance have a parent handle type. In OpenXR, destroying a parent handle also destroys all handle that come from it. For example, if you destroy a Session, that also destroys the associated spaces and swapchains. If you destroy the Instance, that destroys all of the handles shown here, since they all come from the Instance.


Atoms



In addition to handles, there are two additional types known as atoms. They're not objects, and they don't have a explicit lifetime. They're just coded numbers that represent some fixed thing in the runtime. The one you'll work with most often is a path. An XrPath is a number that corresponds to a string representing a semantic path.

Although these have no explicit lifetime, they only have meaning in the Instance they're retrieved from. Do not save or reuse these between runs.



Modeling Interaction with Actions

- **Focus first on *what* users do, not *the hardware* they do it with**
- **Important for hardware-independence.**
- **Action: A semantic (meaningful) bit of interaction**
  - Types: Boolean (button), Float (analog), Vec2, Pose (tracked object), Haptic
  - e.g. “grab_object”, “teleport”, “hand_pose”
- **ActionSet: a group of related actions for a context, environment, etc.**
  - e.g. “menu”, “gameplay”, “driving”
  - One or more active at any time

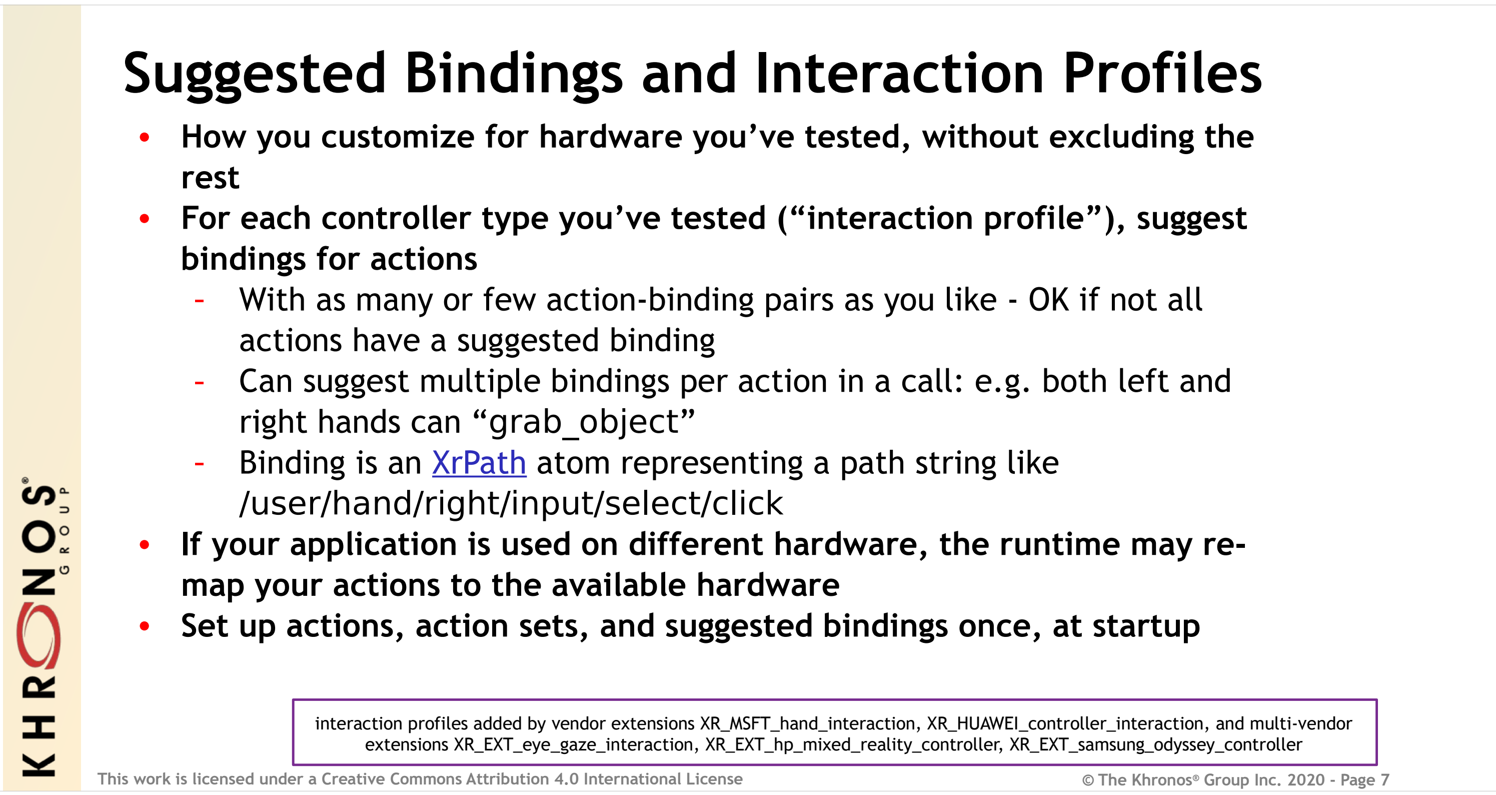
An important part of immersive technology is interaction. Before we get into the details of how an OpenXR application is structured, let’s take a few minutes to look at how OpenXR models interaction, since it might be a new approach for you. OpenXR focuses on the actions that a user takes in your application instead of on the buttons and controllers that are used to perform those actions. This is an important part of the hardware independence provided by OpenXR.

In OpenXR, an Action is a semantic or meaningful bit of interaction: it's something that you do. "Grab object" and "Teleport" are examples of names for actions.

Actions that should be available in a given context or environment are grouped into Action Sets. For example, you might have an Action Set called "menu", one called "gameplay", and one called "driving". You can have one or more Actions sets active at a single time, so you can have "gameplay" and "driving" both active. You do not need to duplicate Actions between sets or create additional Action Sets combining others.

There are several different types of Actions available.

- Boolean is essentially a button action: it has either on or off.
- Float actions are things like an analog trigger.
- Vector2 are two-dimensional floats: things like a thumb stick or trackpad.
- Pose is a special kind of action that's a tracked object. Frequently, these represent hands.
- Finally, haptic actions are an output action allowing you to provide rumble or tactile feedback to the user.



Suggested Bindings and Interaction Profiles

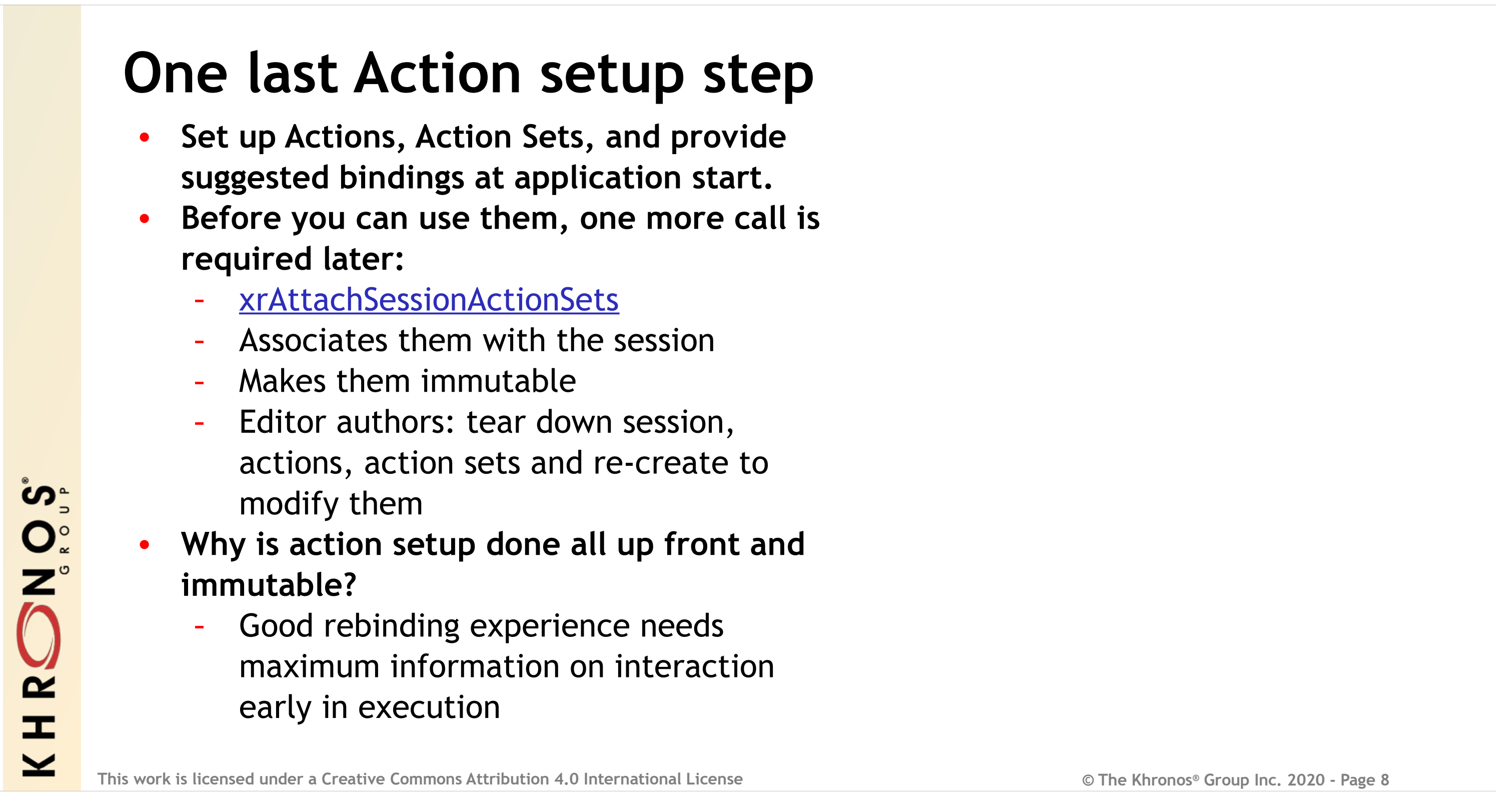
- **How you customize for hardware you've tested, without excluding the rest**
- **For each controller type you've tested ("interaction profile"), suggest bindings for actions**
  - With as many or few action-binding pairs as you like - OK if not all actions have a suggested binding
  - Can suggest multiple bindings per action in a call: e.g. both left and right hands can "grab_object"
  - Binding is an XrPath atom representing a path string like /user/hand/right/input/select/click
- **If your application is used on different hardware, the runtime may re-map your actions to the available hardware**
- **Set up actions, action sets, and suggested bindings once, at startup**

interaction profiles added by vendor extensions XR_MSFT_hand_interaction, XR_HUAWEI_controller_interaction, and multi-vendor extensions XR_EXT_eye_gaze_interaction, XR_EXT_hp_mixed_reality_controller, XR_EXT_samsung_odyssey_controller

Once you've established which logical Actions a user of your application will make, you can then customize how they actually perform those Actions for the hardware that you're testing. This is done using suggested bindings for interaction profiles. For each controller type that you've tested, look up its interaction profile. These are listed in the specification, and cover a number of well-known devices. Additional interaction profiles will be added through extensions. For each device you test with, you submit Actions and their suggested bindings for the corresponding interaction profile. The suggested bindings are the logical part of the controller that you'd like to use to drive that Action: the specific button, for example.

You can suggest multiple bindings per Action in a call. For instance both your left and right hands, and thus your left and right controller, could both cause Action "grab_object". The binding path that you'd like to suggest is a hierarchical string like /user/hand/right/input/select/click. Here you can see that we are referring to the select button on the right hand's controller, and specifically the "click" operation of that select button. For the last two path levels there are naming conventions and standardized names that are detailed in the specification. All these paths are listed in the interaction profile definitions in the specification.

It's important to only suggest bindings for interaction profiles matching devices that you actually tested. Your application will still be able to run on other runtimes: those runtimes may automatically map your Actions to the controller available. As runtimes continue to advance, you can expect that these rebindings will increase in quality, and can be updated independently from your application. They may also provide an interface for the user to customize how the Actions are mapped to their controller, and perhaps even share these bindings with other users. Remapping or rebinding can also be done to improve comfort or accessibility, to support a wider array of users. This part of the specification design is strongly influenced by the success Valve has found with their similar SteamInput and SteamVR Input Action binding and remapping systems.



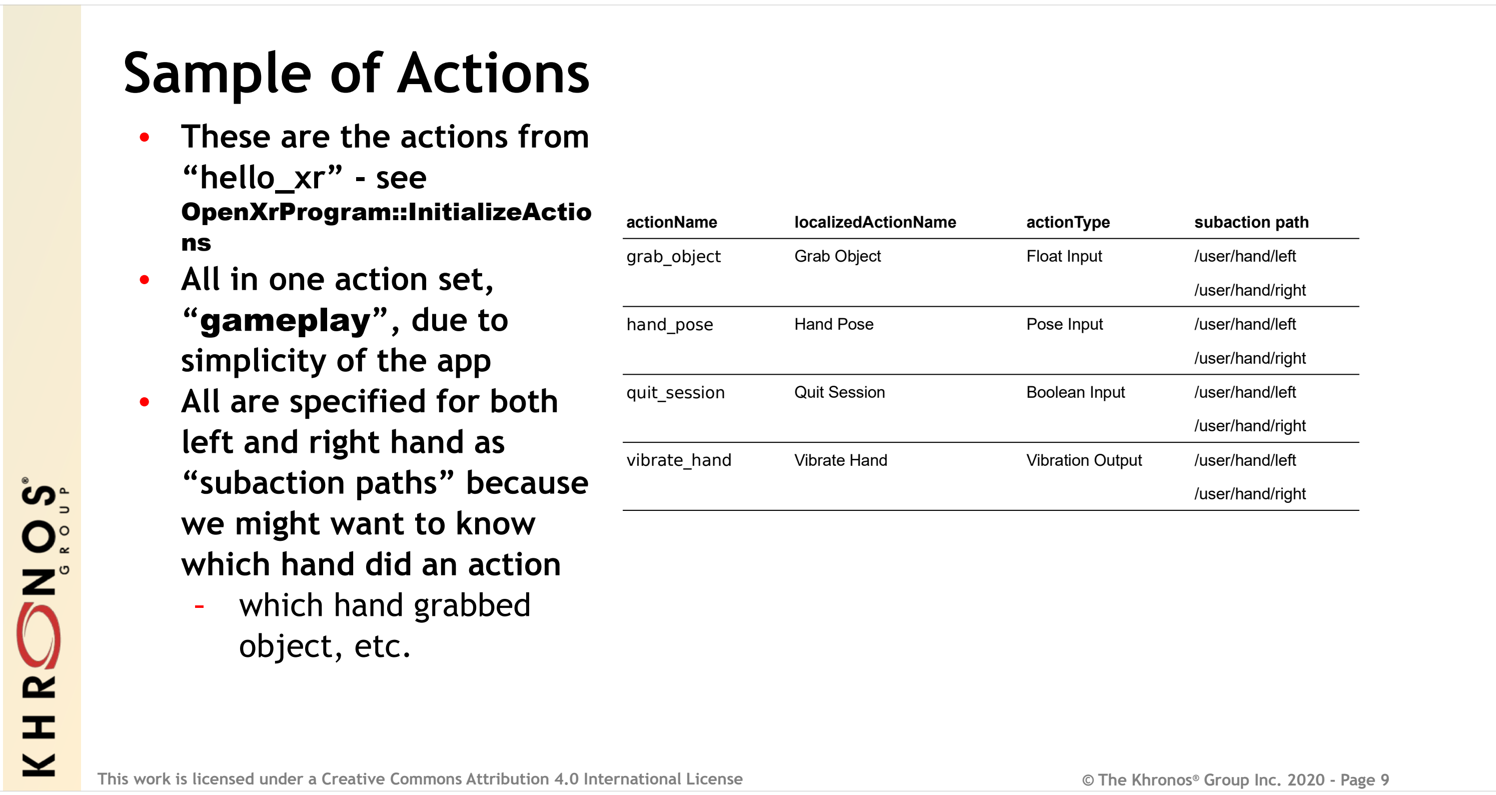
One last Action setup step

- **Set up Actions, Action Sets, and provide suggested bindings at application start.**
- **Before you can use them, one more call is required later:**
  - xrAttachSessionActionSets
  - Associates them with the session
  - Makes them immutable
  - Editor authors: tear down session, actions, action sets and re-create to modify them
- **Why is action setup done all up front and immutable?**
  - Good rebinding experience needs maximum information on interaction early in execution

You will typically set up Actions, Action Sets, and suggest bindings for interaction profiles in a single block of code during the startup process of your application. This model is actually enforced by OpenXR: before you can use actions in your main loop, you need to make a call to "Attach" your Action Sets to a Session. This tells the runtime that I'm all done setting up my Actions, Action Sets, and suggested bindings; I'm going to use them with this Session; and I'm not going to change them anymore. This does make your Actions and Action Sets immutable: you can't modify them after this point and there's a special error code that you'd get if you tried to do that. If you happen to be writing an editor for a game engine, the solution for editing Actions is that each time, you need to tear down the Session, the Actions, and the Action Sets, and then recreate them in order to modify them.

This seems like a pain but there's an important reason that the Action setup is done all up front. The main reason is rebinding. We want the runtime to be able to provide the user with the maximum ability to configure how they're interacting with their application right away. To do this, your application must provide the runtime with the maximum information about its interactions as early as it can. That way, when a user launches your application, if you do not have suggested bindings for the hardware that they have, the runtime can remap the actions automatically, or pop up a UI that lets them map your specified Actions to the hardware that they have available. This would happen behind the scenes and goes unnoticed by your application: you just get compatibility.

If you were able to add Actions and Action Sets later on during execution, not up front, it would then interrupt the flow of your application if rebinding needed to be done a second time. Additionally, if a runtime supports sharing bindings between users, you'd be able to compose a binding that supports all Actions only if Action setup is all done at once. Otherwise, for example, if there's an Action or Action Set that's only used in the last scene or two of your game, then a community created rebinding might be incomplete if that scene was not yet reached or was on a path that wasn't reached, making it less useful in general.



Sample of Actions

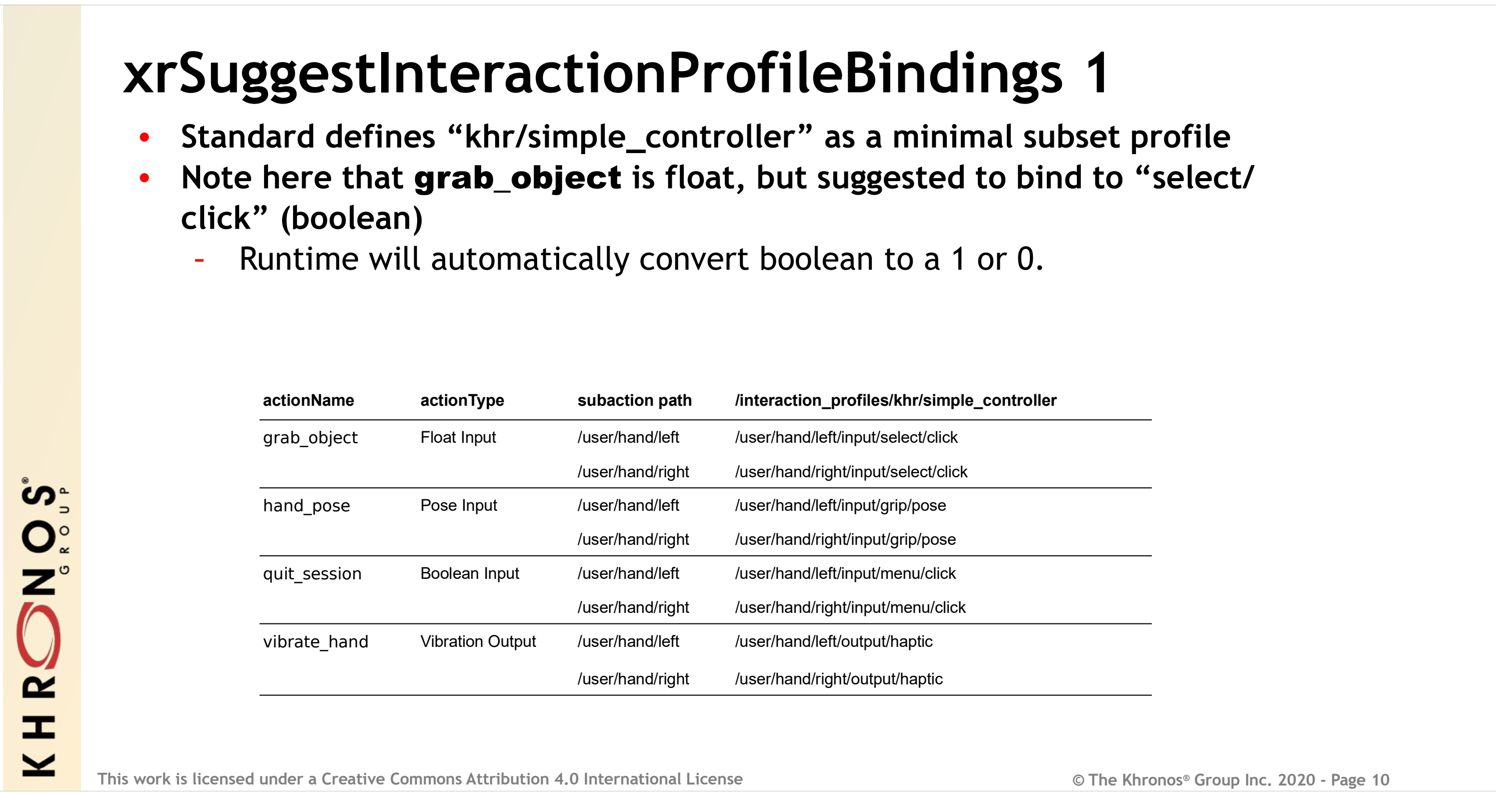
- These are the actions from "hello_xr" - see **OpenXrProgram::InitializeActions**
- All in one action set, "**gameplay**", due to simplicity of the app
- All are specified for both left and right hand as "subaction paths" because we might want to know which hand did an action
  - which hand grabbed object, etc.

actionName	localizedActionName	actionType	subaction path
grab_object	Grab Object	Float Input	/user/hand/left /user/hand/right
hand_pose	Hand Pose	Pose Input	/user/hand/left /user/hand/right
quit_session	Quit Session	Boolean Input	/user/hand/left /user/hand/right
vibrate_hand	Vibrate Hand	Vibration Output	/user/hand/left /user/hand/right

To make this a bit more concrete, I've gone through the sample hello_xr application that's in the OpenXR-SDK-Source and summarized the Actions and bindings that are used there. The InitializeActions member function is where these get set up, if you want to look at the source code on your own later. All these Actions are in a single Action Set because the application is very simple. Additionally, all of these Actions are specified for both the left and right hands. This uses the concept of sub-action paths: this lets the user both perform an Action with either of two hands, but also let you know which one hand actually did it. It's similar to the SteamVR Input concept of restrictToDevice, in case you're familiar with that.

The four Actions represent a variety of Action types. We have

- grab_object
- hand_pose
- quit_session
- vibrate_hand



xrSuggestInteractionProfileBindings 1

- Standard defines “khr/simple_controller” as a minimal subset profile
- Note here that **grab_object** is float, but suggested to bind to “select/click” (boolean)
  - Runtime will automatically convert boolean to a 1 or 0.

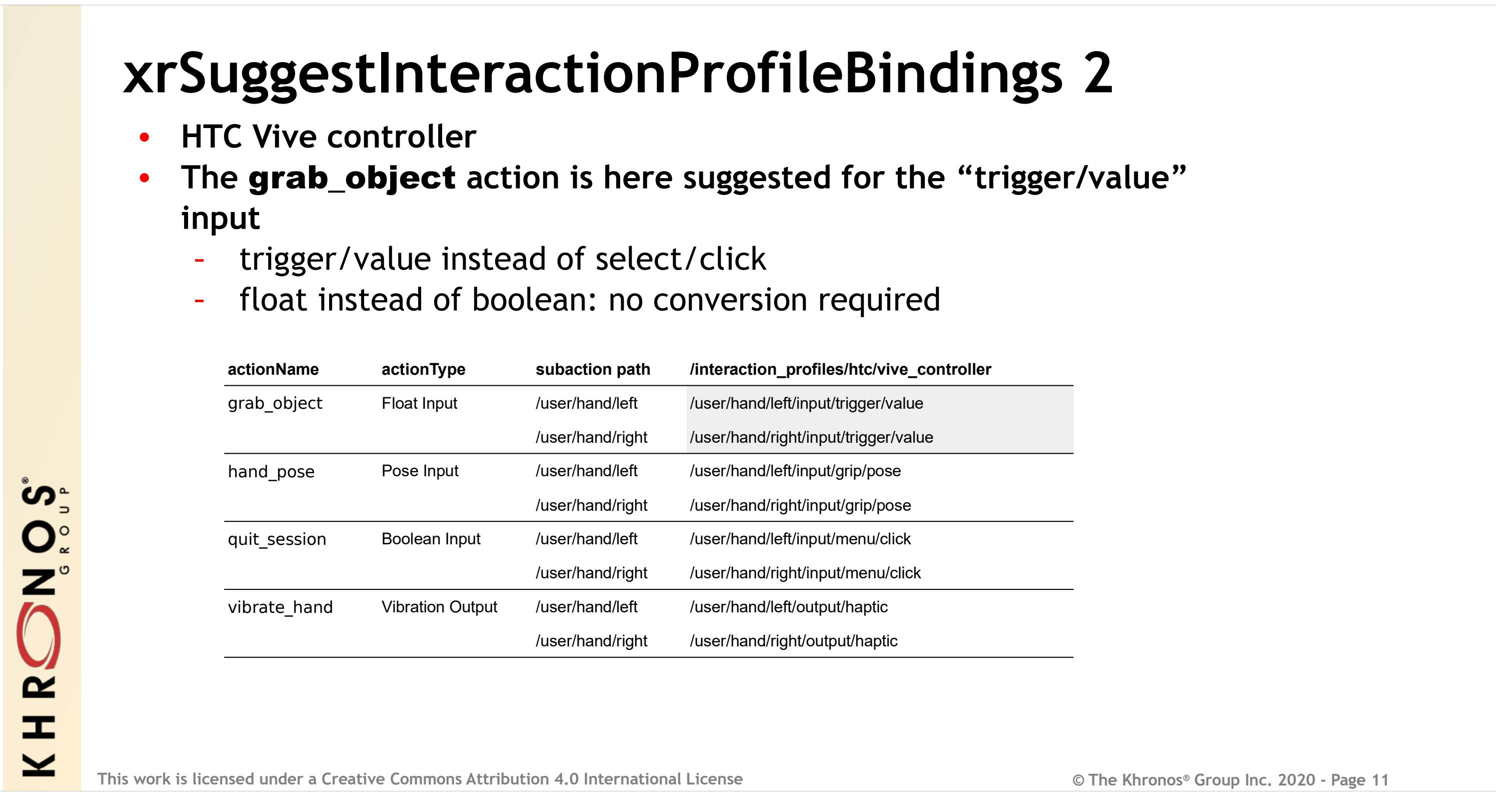
actionName	actionType	subaction path	/interaction_profiles/khr/simple_controller
grab_object	Float Input	/user/hand/left	/user/hand/left/input/select/click
		/user/hand/right	/user/hand/right/input/select/click
hand_pose	Pose Input	/user/hand/left	/user/hand/left/input/grip/pose
		/user/hand/right	/user/hand/right/input/grip/pose
quit_session	Boolean Input	/user/hand/left	/user/hand/left/input/menu/click
		/user/hand/right	/user/hand/right/input/menu/click
vibrate_hand	Vibration Output	/user/hand/left	/user/hand/left/output/haptic
		/user/hand/right	/user/hand/right/output/haptic

After creating that action set and those four actions, it's time to suggest bindings. There are a number of calls in hello_xr to suggest bindings. I've picked a few of them for three different interaction profiles to illustrate some points.

This first one uses the interaction profile khr/simple_controller. Unlike most other interaction profiles, this does not correspond to any particular specific piece of hardware. Instead, it's a generic lowest-common-denominator sort of device that can be mapped to a wide variety of hardware.

There are a few things to notice here. Overall, as you'll see is a pattern, the suggested binding path for the interaction profile starting with /user/hand/left will be marked as for the subaction path of /user/hand/left, and similarly similarly for /user/hand/right. In this application all Actions can be done with either hand, but you would not see this same pattern if you had Actions that could only be performed by one hand.

One point about the simple controller is that its select input is boolean: it only has on or off. We're binding grab_object, which is a float input, to /user/hand/left/input/select/click, which is boolean. This is fine: the runtime will automatically convert that boolean value to a float one or zero. There are conversion rules that are described in the specification for these common, simple cases.



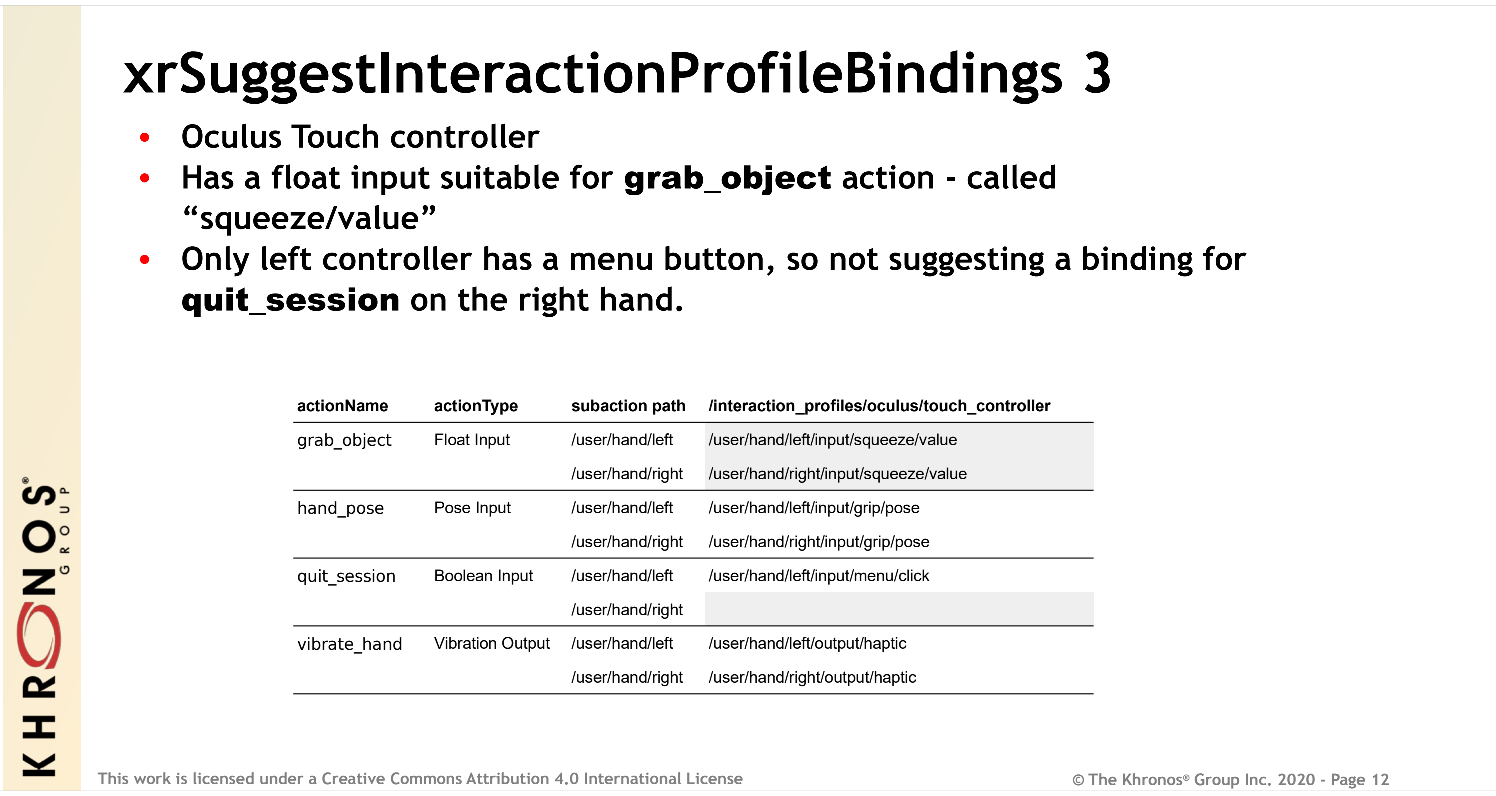
xrSuggestInteractionProfileBindings 2

- HTC Vive controller
- The **grab_object** action is here suggested for the “trigger/value” input
  - trigger/value instead of select/click
  - float instead of boolean: no conversion required

actionName	actionType	subaction path	/interaction_profiles/htc/vive_controller
grab_object	Float Input	/user/hand/left	/user/hand/left/input/trigger/value
		/user/hand/right	/user/hand/right/input/trigger/value
hand_pose	Pose Input	/user/hand/left	/user/hand/left/input/grip/pose
		/user/hand/right	/user/hand/right/input/grip/pose
quit_session	Boolean Input	/user/hand/left	/user/hand/left/input/menu/click
		/user/hand/right	/user/hand/right/input/menu/click
vibrate_hand	Vibration Output	/user/hand/left	/user/hand/left/output/haptic
		/user/hand/right	/user/hand/right/output/haptic

KHRONOS® GROUP

The second example is the HTC Vive controller. Here you can see that we've bound the grab_object action to a different path. That's because the Vive controller has an analog trigger rather than a button simply labeled "Select". This analog trigger, which is a float, is now being suggested as our binding for grab object. The runtime will not need to convert a boolean to a float. The grab_object action will get something that's not just limited to 0 or 1 but might be anywhere in that entire range.

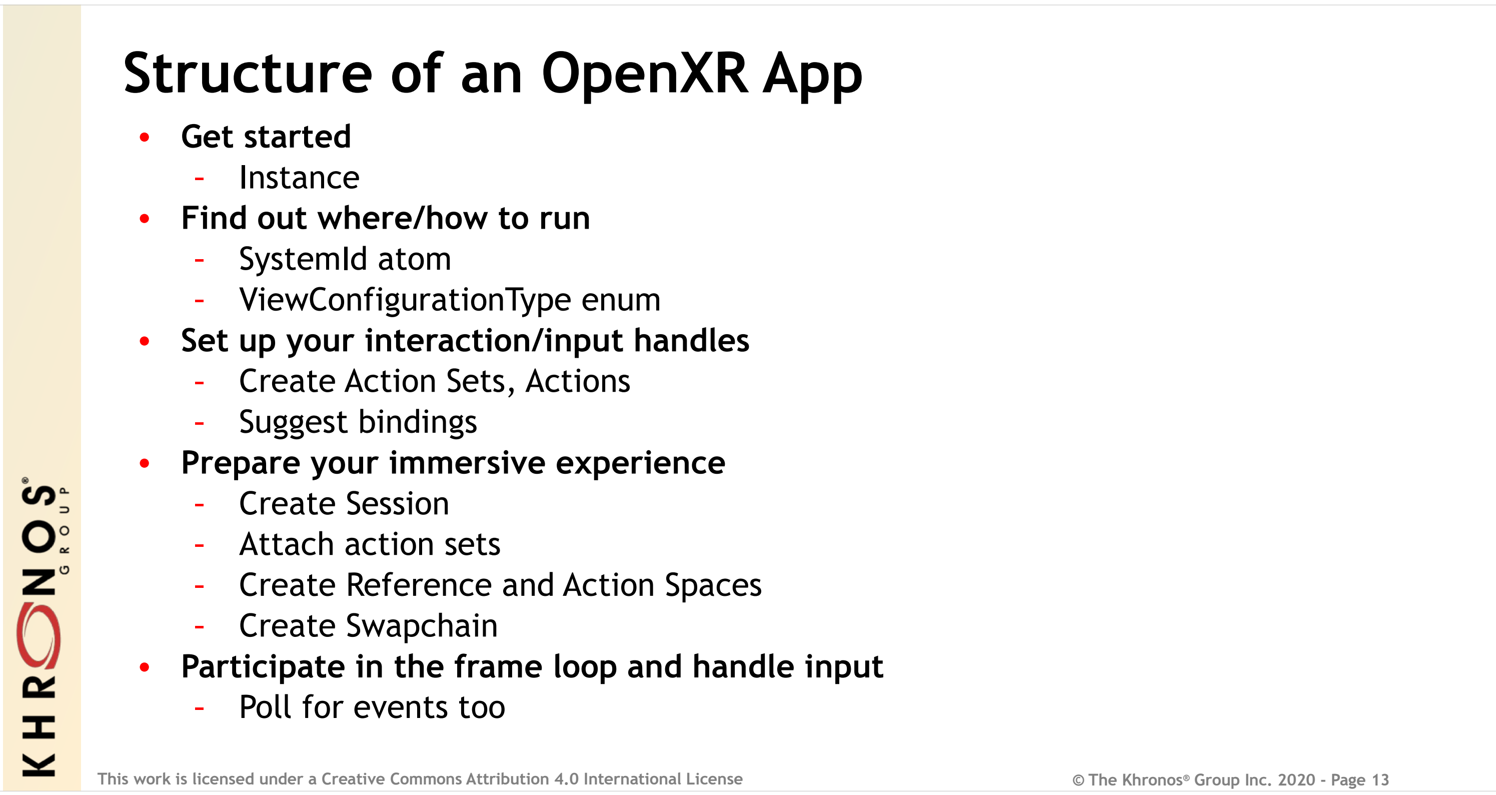


xrSuggestInteractionProfileBindings 3

- Oculus Touch controller
- Has a float input suitable for **grab_object** action - called “squeeze/value”
- Only left controller has a menu button, so not suggesting a binding for **quit_session** on the right hand.

actionName	actionType	subaction path	/interaction_profiles/oculus/touch_controller
grab_object	Float Input	/user/hand/left	/user/hand/left/input/squeeze/value
		/user/hand/right	/user/hand/right/input/squeeze/value
hand_pose	Pose Input	/user/hand/left	/user/hand/left/input/grip/pose
		/user/hand/right	/user/hand/right/input/grip/pose
quit_session	Boolean Input	/user/hand/left	/user/hand/left/input/menu/click
		/user/hand/right	
vibrate_hand	Vibration Output	/user/hand/left	/user/hand/left/output/haptic
		/user/hand/right	/user/hand/right/output/haptic

The last example of suggested bindings is the Oculus Touch controller. The Oculus Touch controller has a float squeeze input: it can report how hard you are squeezing using a floating point value between 0 & 1. The grab_object action will get a value anywhere in that range, just like the analog trigger on the Vive controller. Additionally, only the left controller has a menu button, so in this case we're only suggesting a binding for the left hand for quit_session which uses the menu button. We're not suggesting a binding for /user/hand/right and that's okay.



Structure of an OpenXR App

- **Get started**
  - Instance
- **Find out where/how to run**
  - SystemId atom
  - ViewConfigurationType enum
- **Set up your interaction/input handles**
  - Create Action Sets, Actions
  - Suggest bindings
- **Prepare your immersive experience**
  - Create Session
  - Attach action sets
  - Create Reference and Action Spaces
  - Create Swapchain
- **Participate in the frame loop and handle input**
  - Poll for events too

This is the structure of an OpenXR app. We'll go into all of these in more detail.

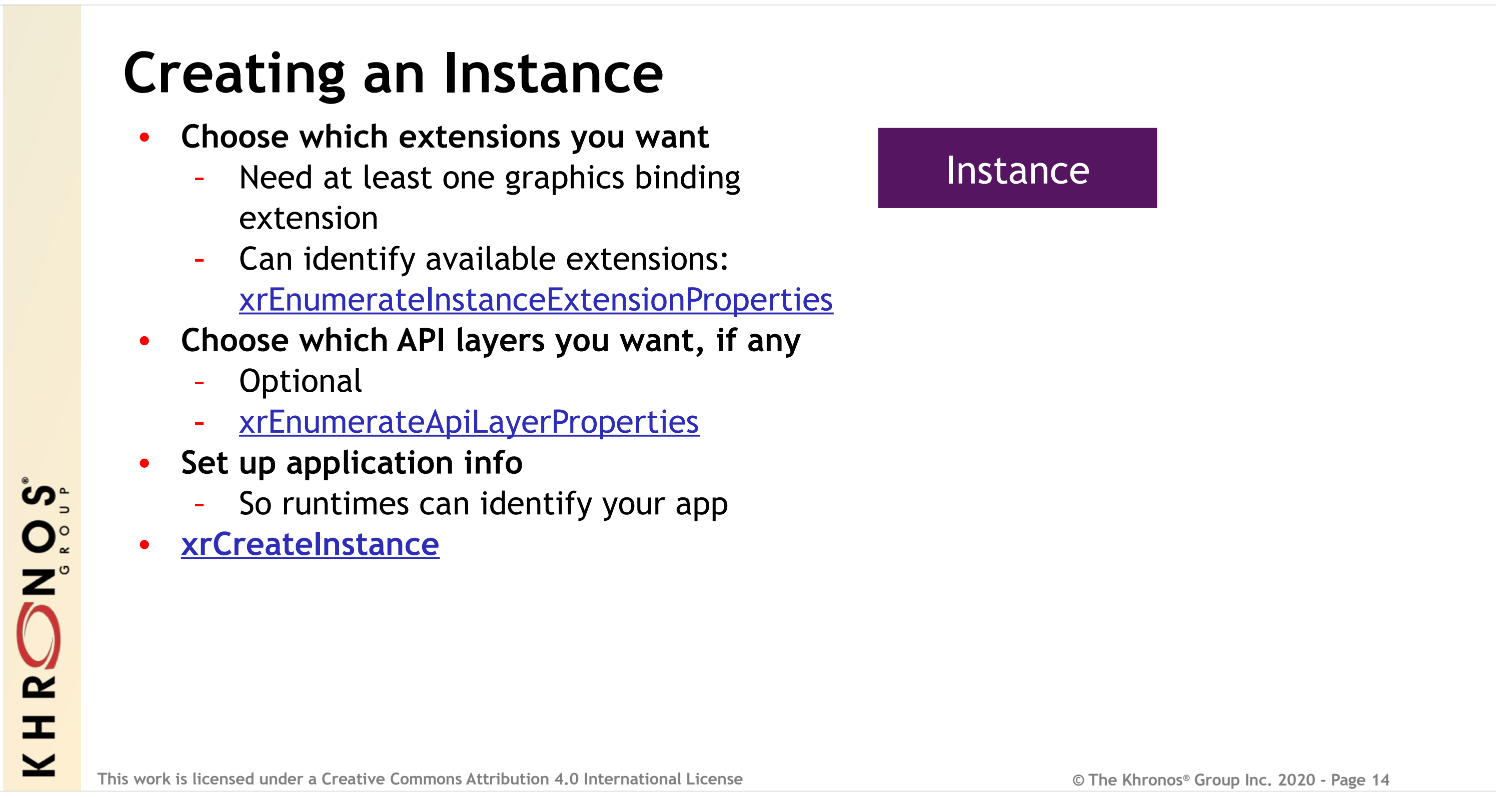
First, get started by configuring and creating your instance.

Next, you find out where and how to run: this involves looking up a system ID atom and using the view configuration type enum.

Then, you set up your interaction in OpenXR. This is done using action sets and actions.

Finally, the last setup step is preparing the immersive experience by creating your session, attaching action sets, creating reference and action spaces, and creating your swap chain.

The body of your application consists of advancing the frame loop, responding to input, and handling events.



Creating an Instance

- Choose which extensions you want
  - Need at least one graphics binding extension
  - Can identify available extensions: xrEnumerateInstanceExtensionProperties
- Choose which API layers you want, if any
  - Optional
  - xrEnumerateApiLayerProperties
- Set up application info
  - So runtimes can identify your app
- xrCreateInstance

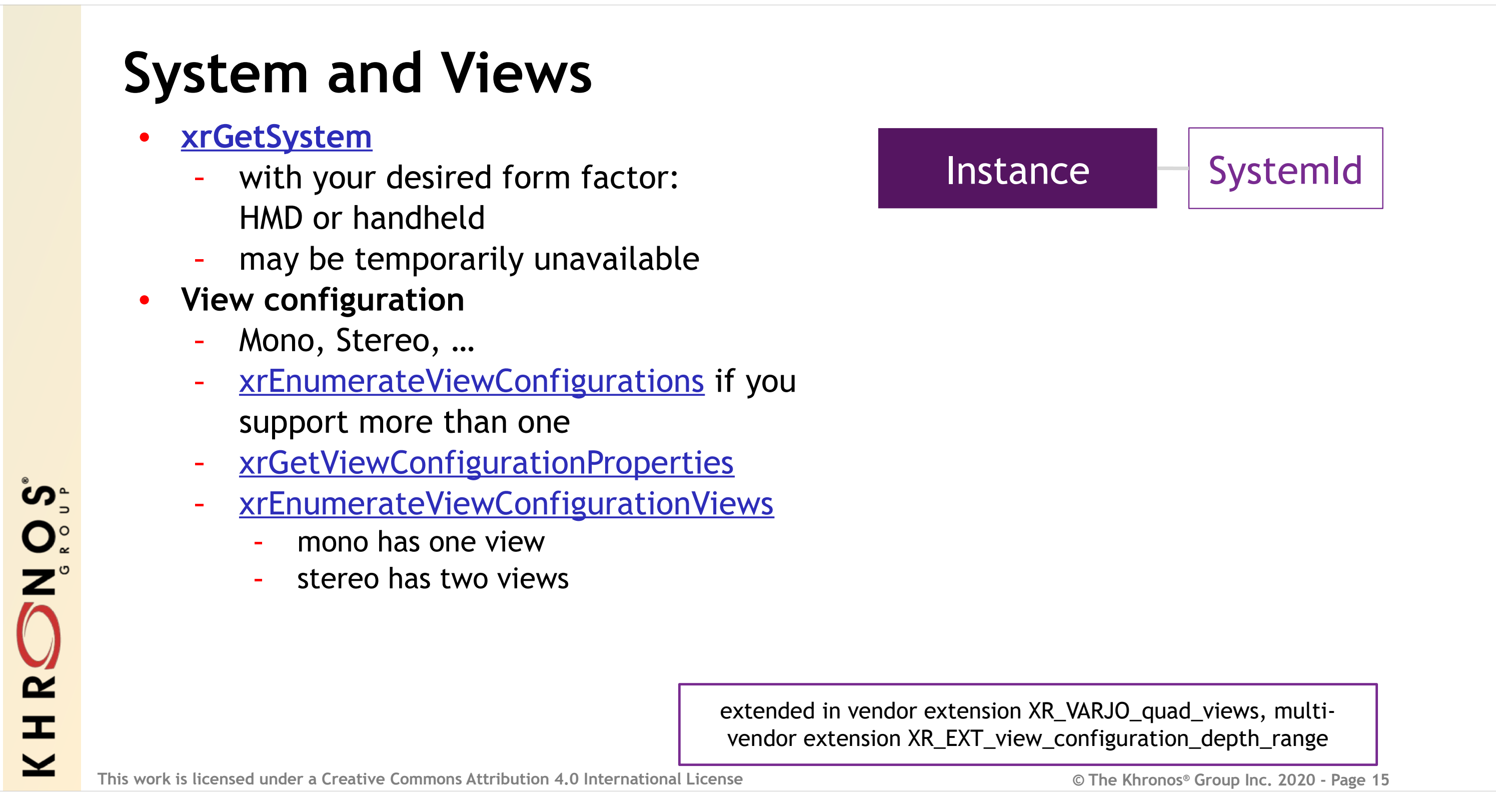
Instance

When you create an instance, you first need to choose which extensions you want, very similar to how Vulkan works. In order to make an OpenXR application you need at least one extension enabled, and that's a graphics binding extension. You can determine which extensions are all available on the system that you're using by using xrEnumerateInstanceExtensionProperties. However, if you don't have any optional extension usage, only required extensions, you can just proceed to create the instance and ask for the extensions you need: it will either succeed if they are all available, or return an error if one or more is not available.

There's also a facility for API layers as I mentioned earlier. These can be configured outside of your application through environment variables or similar to make the loader automatically load them. However if your application wants to load them explicitly you can enumerate which ones are available before you create an instance.

Similar to Vulkan, there's an application info struct that you should fill out with your application name and engine name version, so that runtimes can identify your application.

Finally, xrCreateInstance takes that information and hands you an instance handle which you will use in the rest of your app.



System and Views

- xrGetSystem
  - with your desired form factor: HMD or handheld
  - may be temporarily unavailable
- **View configuration**
  - Mono, Stereo, …
  - xrEnumerateViewConfigurations if you support more than one
  - xrGetViewConfigurationProperties
  - xrEnumerateViewConfigurationViews
    - mono has one view
    - stereo has two views

Instance

SystemId

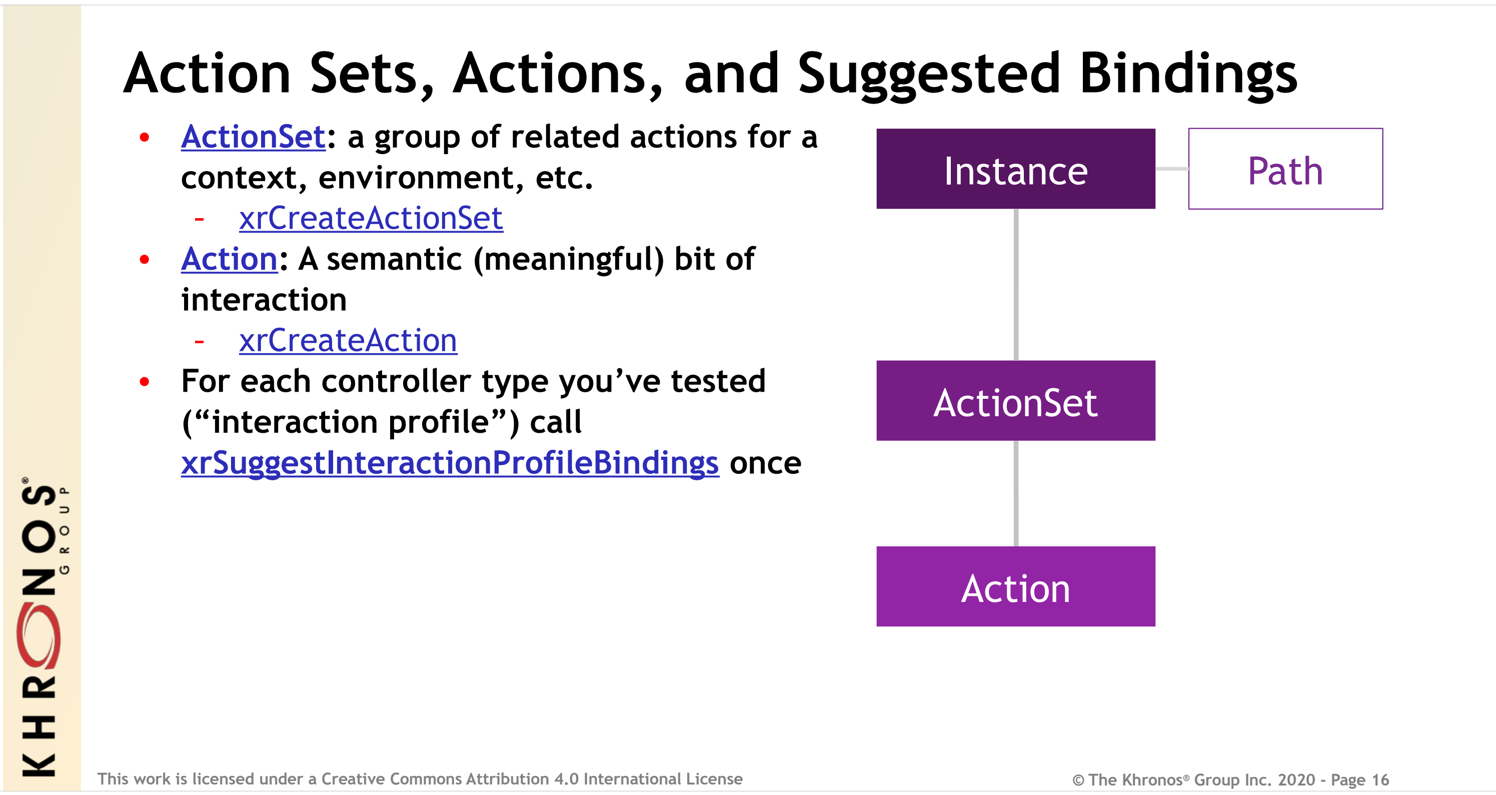
extended in vendor extension XR_VARJO_quad_views, multi-vendor extension XR_EXT_view_configuration_depth_range

Use xrGetSystem to find your desired form factor. OpenXR 1.0 natively supports without extensions stereo head-mounted displays as well as handheld mono "magic window" style augmented reality. However, not all runtimes will support both of these form factors of devices. Part of the app startup process is asking if there is a system of the form factor you wish available.

There are a few possible outcomes: the form factor that you asked for might be available, in which case you'd get a system ID. It might be never available if the device that you're using can't do the form factor that you asked for. Or, finally, it might be temporarily unavailable, if it's perhaps not plugged in or if it needs to transition to a different mode in order to be used in that way. For example, a phone with a slide-in VR headset is a device that can be used in multiple form factors, and stereo head-mounted display would be unavailable when not in the headset.

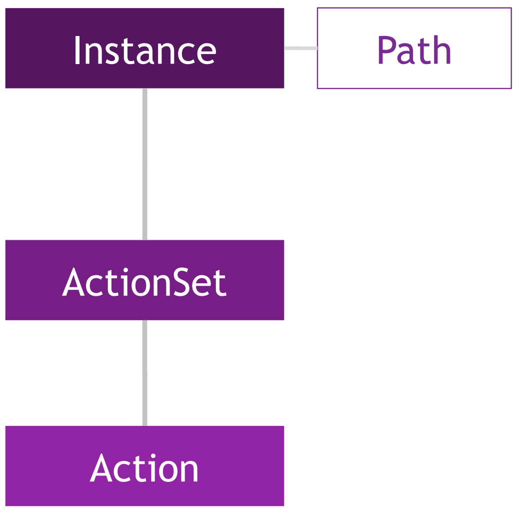
Once you have a system, you set up your view configuration. This is where mono, stereo, or even more views come in. If you support more than one view configuration, you can use xrEnumerateViewConfigurations to find out which ones are supported by the system you've chosen, to make your determination of which one to use.

No matter which view configuration you use, you will then call xrEnumerateViewConfigurationViews to get the details of each of the views for your view configuration. Each view configuration has a known number of views: mono has one, stereo has two, the Varjo quad view provided by an extension has four, and so on. These are well known and listed in the specification.



Action Sets, Actions, and Suggested Bindings

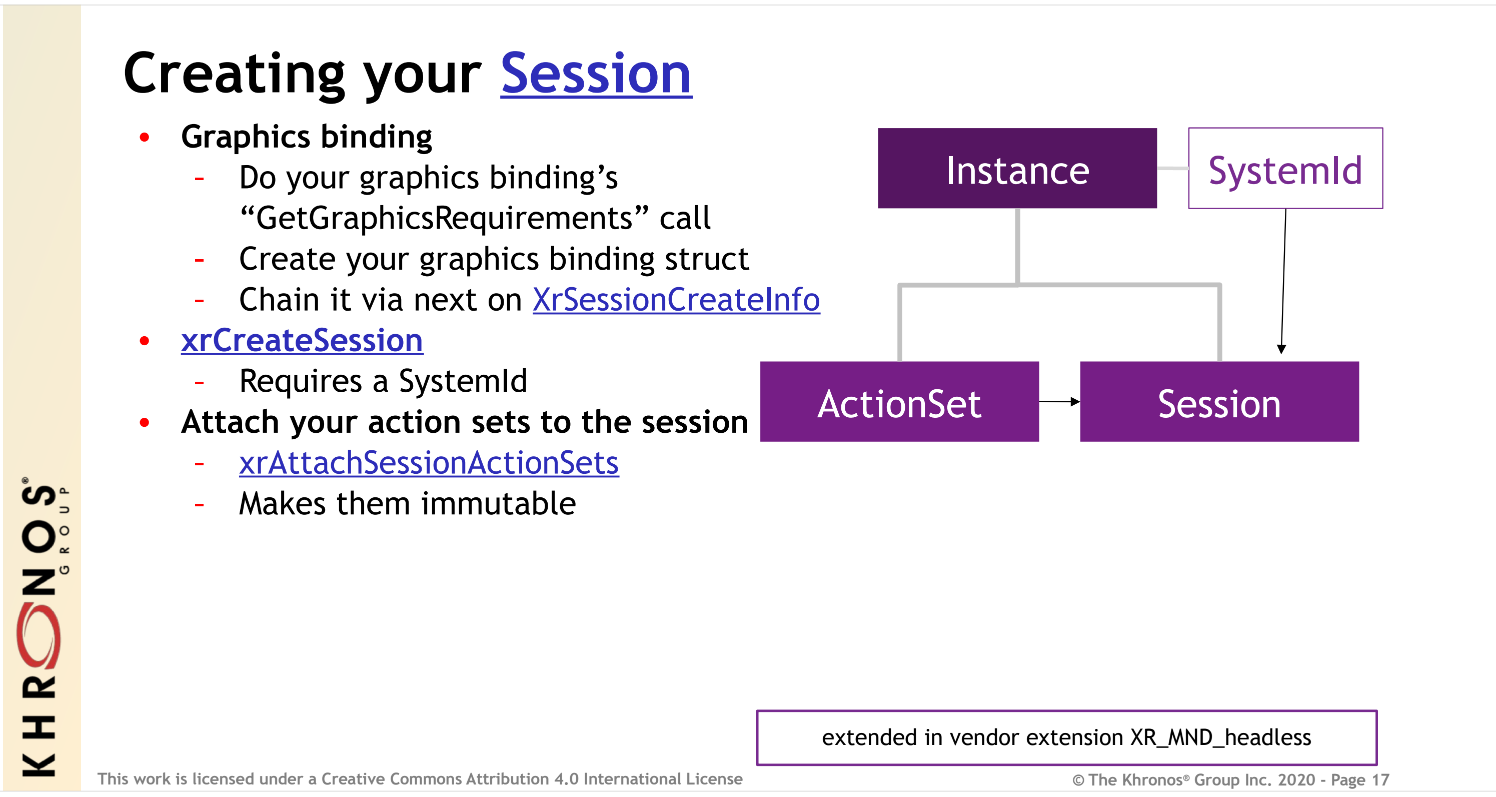
- ActionSet: a group of related actions for a context, environment, etc.
  - xrCreateActionSet
- Action: A semantic (meaningful) bit of interaction
  - xrCreateAction
- For each controller type you've tested ("interaction profile") call xrSuggestInteractionProfileBindings once



Once you have an Instance, you can get your interactions set up: these are your Action Sets and Actions. We have already discussed most details of action setup. You create an Action Set with xrCreateActionSet. You create all Actions, no matter the type, using xrCreateAction.

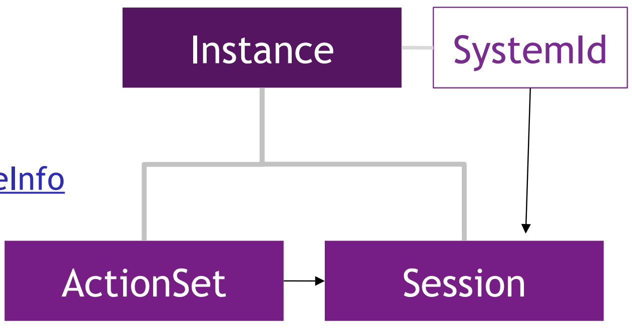
After creating all Action Sets and Actions, you can suggest bindings. Call xrSuggestInteractionProfileBindings once for each controller type, or "interaction profile", you've tested, passing as many action-bindings as you like. If you call xrSuggestInteractionProfileBindings more than once for a single interaction profile, only your last successful call will be considered by the runtime.

Note that Actions and Action Sets are children of Instance, but they can only be used when attached to a Session.



Creating your Session

- Graphics binding
  - Do your graphics binding's "GetGraphicsRequirements" call
  - Create your graphics binding struct
  - Chain it via next on XrSessionCreateInfo
- xrCreateSession
  - Requires a SystemId
- Attach your action sets to the session
  - xrAttachSessionActionSets
  - Makes them immutable



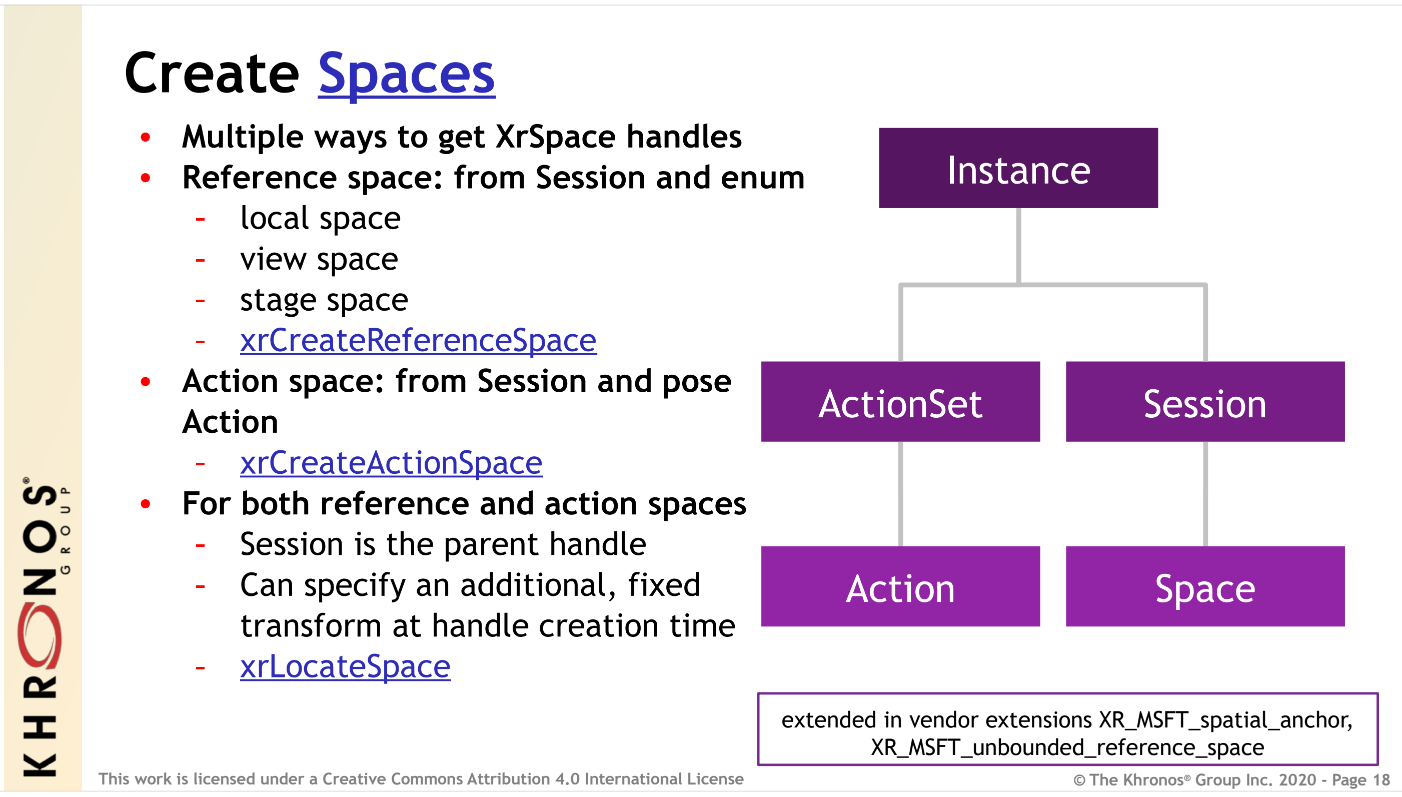
extended in vendor extension XR_MND_headless

The next major handle to create is your Session. You'll first want to get your graphics binding ready. Depending on which graphics API you use there will be a "GetApiGraphicsRequirements" call that's specified by that extension. You need to call that before calling xrCreateSession, or you'll get an error. This provides useful information on how to configure your rendering in order to get the rendered content onto the display.

You then create a graphics binding struct. These are also specified by the graphics binding extensions, and they're all chained on via the "next" pointer on XrSessionCreateInfo. I haven't talked about the "next" pointer too much, but similar to Vulkan, OpenXR structures contain a "type" field as well as a void pointer field named "next" that allows you to add additional structures to the parameters of a call as a "chain". This is mostly used for extension functionality, but there are a few cases in the core specification where using the "next" pointer is required, and this is one of them.

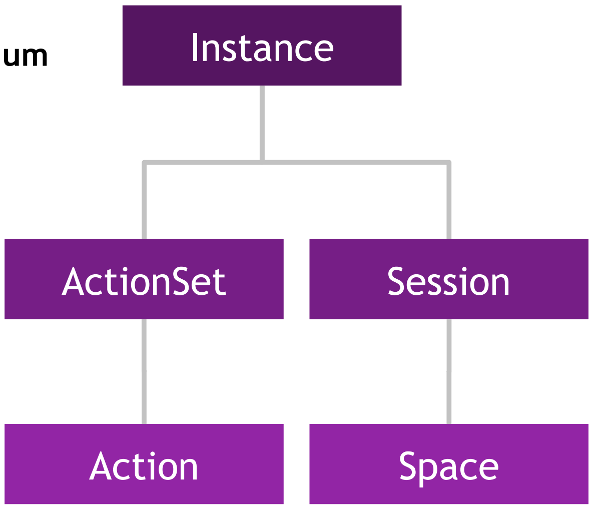
In addition to the graphics binding struct, you also must provide your SystemID from earlier.

Once you have that Session, you then need to attach your Action Sets to it by calling xrAttachSessionActionSets. As we discussed earlier, this makes your actions, action sets, and suggested bindings immutable.



Create Spaces

- Multiple ways to get XrSpace handles
- Reference space: from Session and enum
  - local space
  - view space
  - stage space
  - xrCreateReferenceSpace
- Action space: from Session and pose Action
  - xrCreateActionSpace
- For both reference and action spaces
  - Session is the parent handle
  - Can specify an additional, fixed transform at handle creation time
  - xrLocateSpace

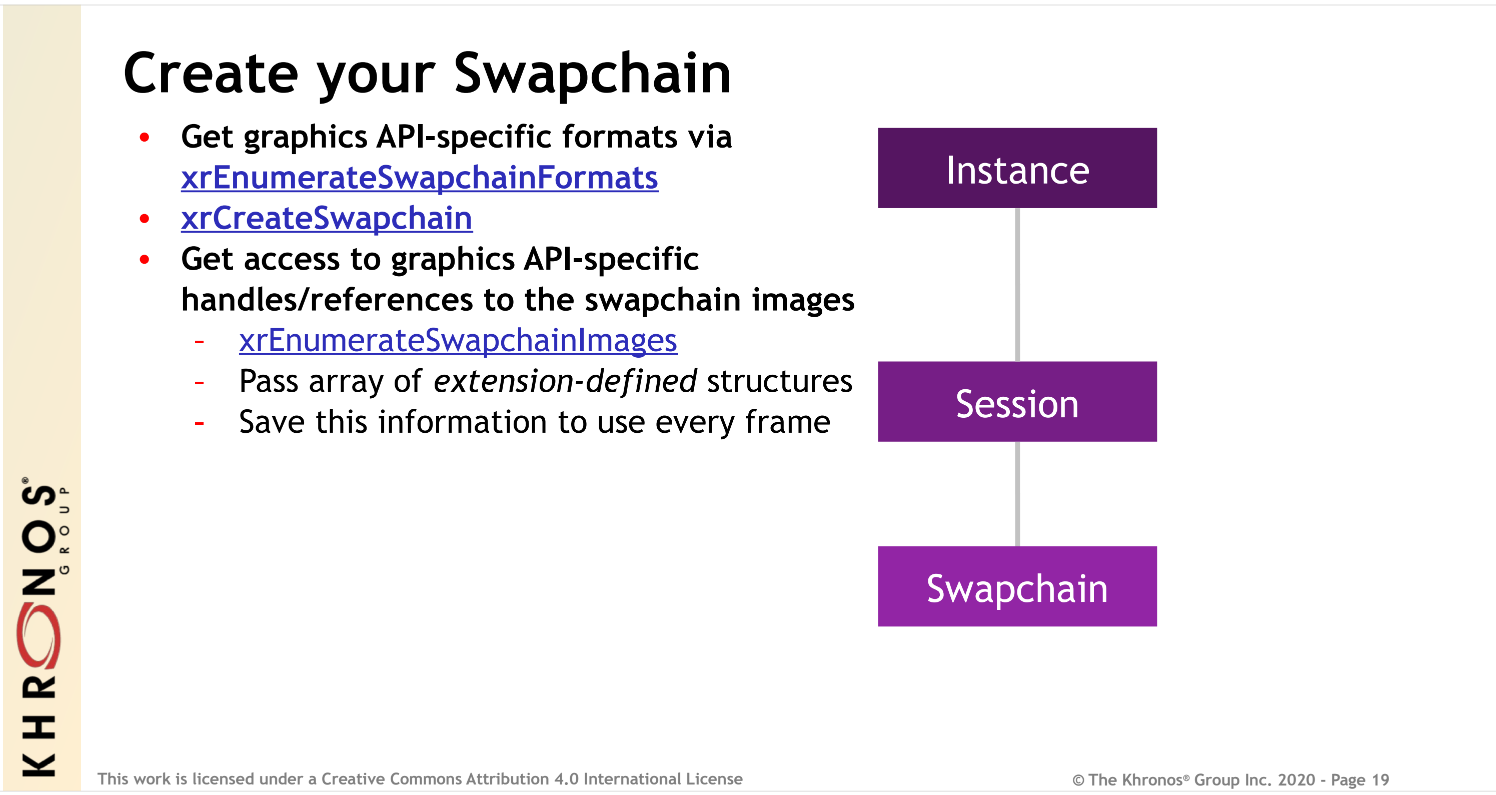


extended in vendor extensions XR_MSFT_spatial_anchor, XR_MSFT_unbounded_reference_space

To interact with tracked objects, you use XrSpace handles. There are multiple ways to get these handles. Several spaces are known as reference spaces: you access these using your XrSession and an enum. Three of these are local space, view space, and stage space. There are additional ones added in extensions. Stage space can be considered a bounded area, standing play environment. Local space is seated play space. View space is essentially head space, for control and rendering something that's head-locked. To get an XrSpace from these enums, you use xrCreateReferenceSpace.

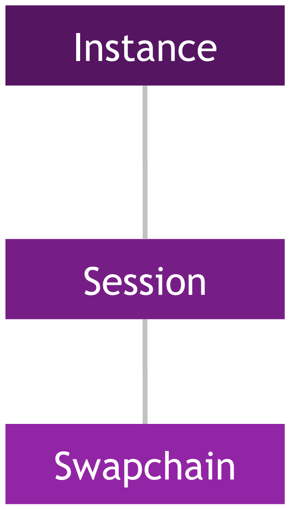
Another kind of space is an Action space. To create these, you use your XrSession and a pose Action. The result is an XrSpace just the same as with CreateReferenceSpace, so your use of them after creation is mostly the same.

For both of these types of spaces, Session is the parent handle. Additionally, for both of these space types, you can specify an additional fixed transform at handle creation time xrLocatSpace is the call used to find the transform from one space to another. Note that you never just find the pose of a space: you always find the location of a space with respect to another space. For example, you don't track your hand; you track your hand relative to local space or stage space.



Create your Swapchain

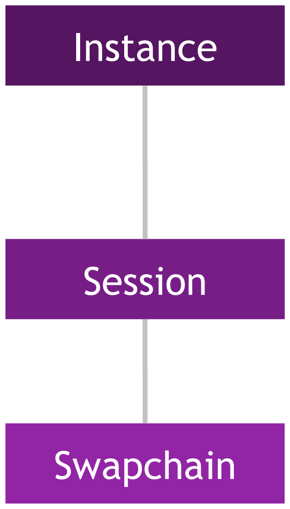
- Get graphics API-specific formats via xrEnumerateSwapchainFormats
- xrCreateSwapchain
- Get access to graphics API-specific handles/references to the swapchain images
  - xrEnumerateSwapchainImages
  - Pass array of *extension-defined* structures
  - Save this information to use every frame



To render, you'll need to create a swapchain. You'll obtain your graphics-API-specific formats through xrEnumerateSwapchainFormats, then you'll call xrCreateSwapchain one or more times. Once you've created a swapchain, you will access the graphics-API-specific handles or references to the swapchain images using xrEnumerateSwapchainImages. This is a slightly unusual call: you'll pass an array of extension defined structures to this core OpenXR function. You'll want to save the information that comes back from this call to use every frame. The contents of those structures specifies in a graphics-API-specific way where to render your image.

Frame loop

- **Frame functions: called on the Session**
  - xrWaitFrame to block until head-pose-dependent sim and rendering
  - xrBeginFrame to mark start of render
  - xrEndFrame to submit the image
  - Populate XrFrameEndInfo::displayTime using output of xrWaitFrame



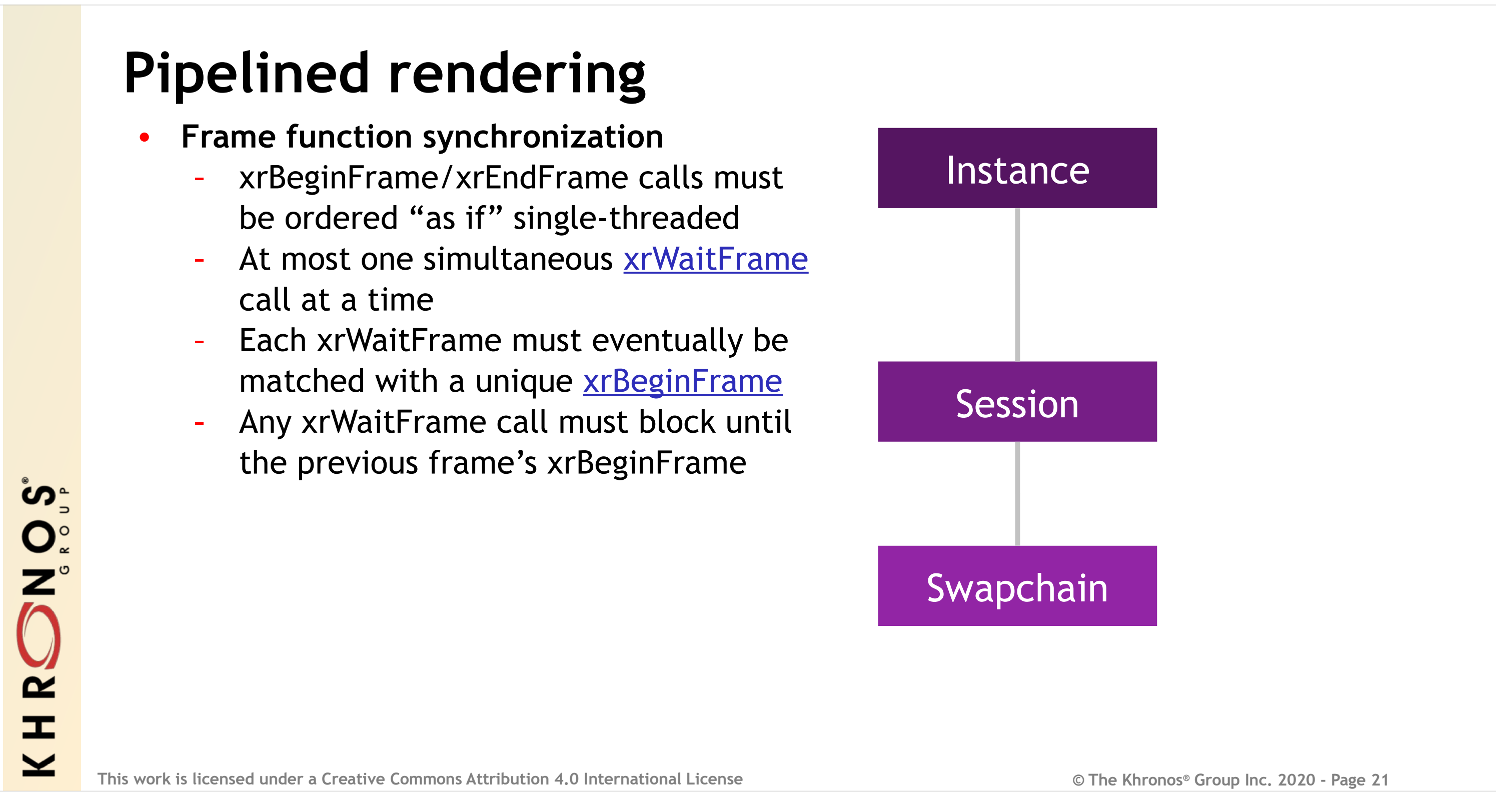
Within the frame loop there are three functions with "Frame" in their name that control the lifecycle of a frame.

xrWaitFrame is a scheduling call. When you call it, it blocks and does not return until the runtime determines you can proceed with head-pose-dependent simulation and rendering. It also provides you with the predicted display time for the frame you're working on rendering. You'll use this time in all your calculations and all your space locations.

xrBeginFrame is executed by your application to mark the start of rendering or GPU usage for that frame.

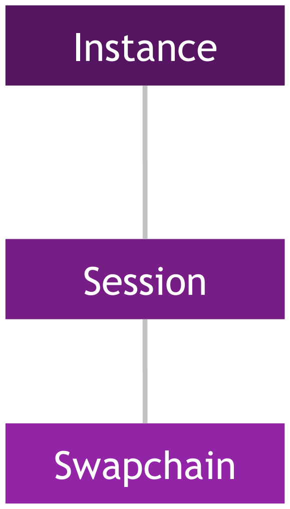
Finally, xrEndFrame submits the frame for display.

You populate the XrEndFrameInfo::displayTime using the predicted display time from xrWaitFrame.



Pipelined rendering

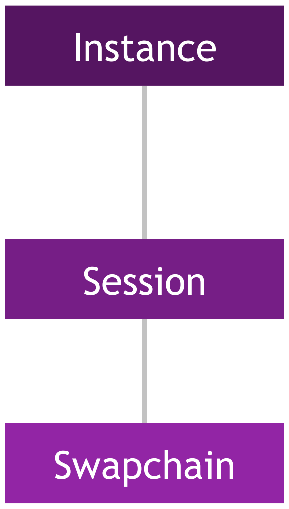
- Frame function synchronization
  - xrBeginFrame/xrEndFrame calls must be ordered "as if" single-threaded
  - At most one simultaneous xrWaitFrame call at a time
  - Each xrWaitFrame must eventually be matched with a unique xrBeginFrame
  - Any xrWaitFrame call must block until the previous frame's xrBeginFrame



If your application is using pipelined or multi-threaded rendering there are some more detailed timing requirements that are important to know. xrBeginFrame and EndFrame calls must be ordered as if they were single threaded, although they may be called from any thread. You can have at most one simultaneous xrWaitFrame call being executed at a time and each xrWaitFrame must eventually be matched with a unique xrBeginFrame. They come in pairs: each WaitFrame has a BeginFrame and every BeginFrame has a WaitFrame. Additionally, any xrWaitFrame call will block in the runtime until the previous frame's xrBeginFrame call has been made.

Swapchain and view management

- **Swapchain management**
  - xrAcquireSwapchainImage to get index
    - To look up/create your command buffers
  - xrWaitSwapchainImage before writing
    - Typically immediately after acquire
    - Do not submit command buffers until this returns
  - xrReleaseSwapchainImage before xrEndFrame: implicitly uses most recently released image
- xrLocateViews



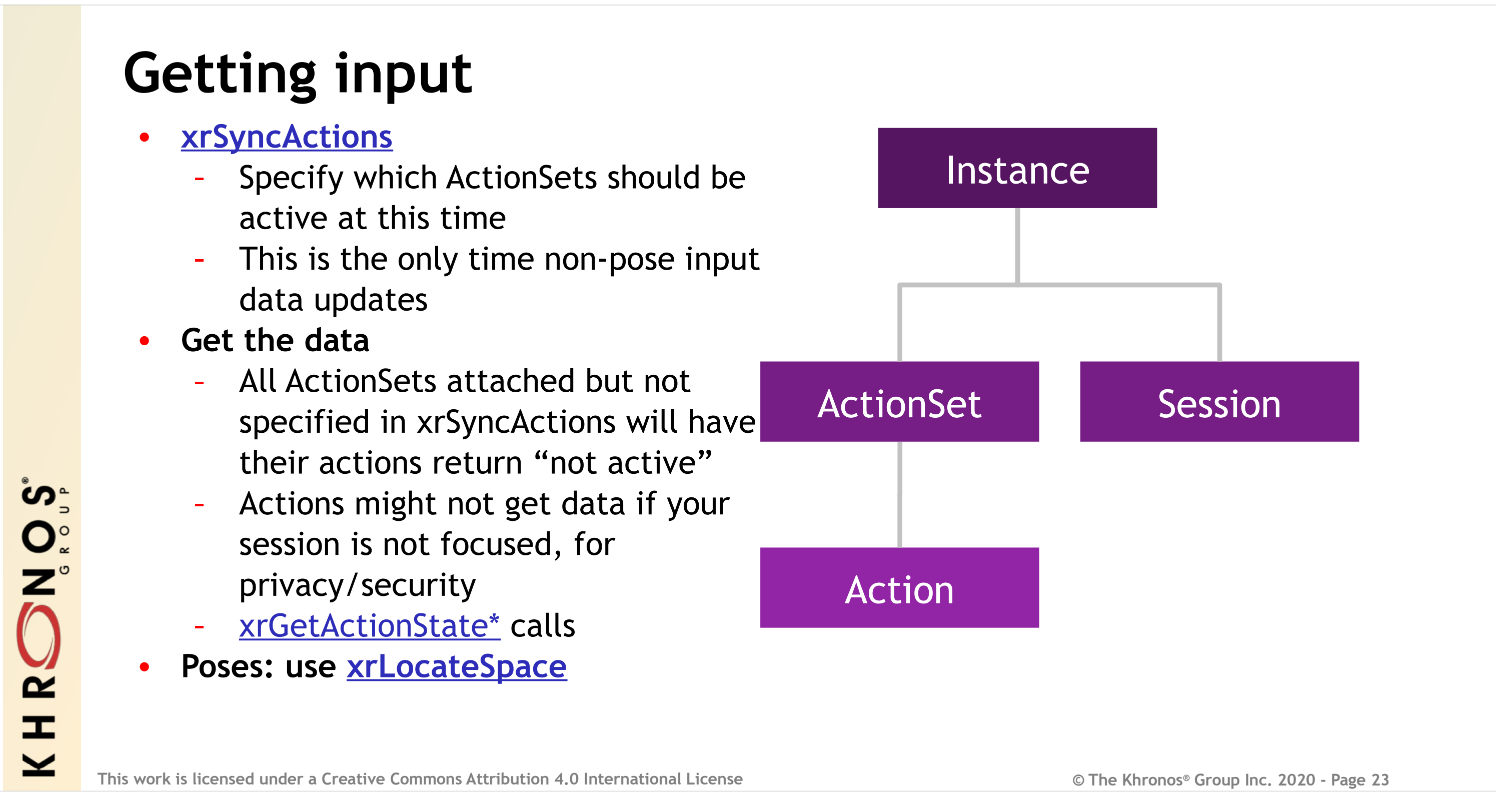
Between begin and end frame, once it's time to actually render, you'll need to use the swapchain that you created earlier.

xrAcquireSwapchainImage does not give you permission to write to the image but it does get the index of the swapchain image you will use this frame. You can use this index to find the graphics-API-specific handle that you enumerated and cached with your xrEnumerateSwapchainImages call earlier.

xrWaitSwapchainImage must be called before writing to that image. It's typically called immediately after Acquire. However, as an optimization, you may lookup or create your command buffers using just the index from AcquireSwapchainImage before waiting for the compositor to release the image for writing by your application.

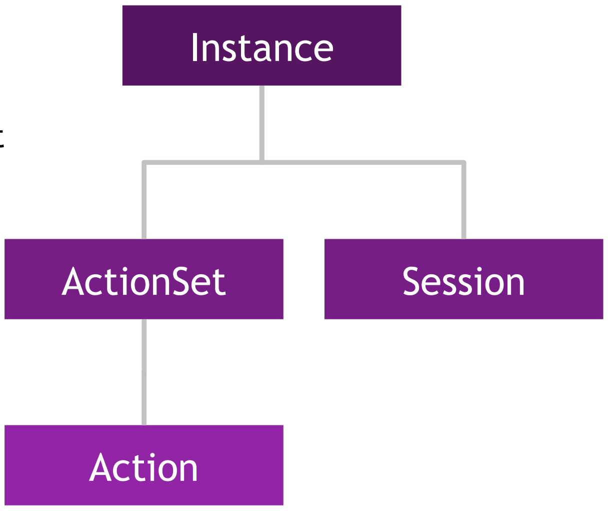
xrReleaseSwapchainImage is what you call when you're all done rendering, right before calling xrEndFrame. xrEndFrame implicitly uses the most recently released swapchain image for displaying to the device.

When you're doing your rendering, you need to render for the predicted display time and for the predicted head pose at that time. xrLocateViews is how you look up that information. It works very similarly to xrLocateSpace.



Getting input

- xrSyncActions
  - Specify which ActionSets should be active at this time
  - This is the only time non-pose input data updates
- **Get the data**
  - All ActionSets attached but not specified in xrSyncActions will have their actions return “not active”
  - Actions might not get data if your session is not focused, for privacy/security
  - xrGetActionState* calls
- **Poses: use xrLocateSpace**



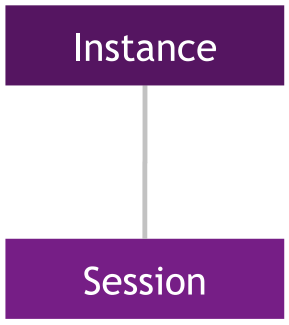
Now that we have an application that can render, we should get input to make it interactive. xrSyncActions should be called once per simulation frame in your application. It specifies which Action Sets should be active for that frame, and updates all non-pose input data in those active Action Sets. Any Action Sets not specified in the most recent xrSyncActions call are considered "not active" and will not get updated data.

After you SyncActions, then get the action data. Actions belonging to an inactive Action Set will return that they are "not active". Additionally, Actions might become inactive and might not get data if your Session is not focused, for privacy and security purposes. To get action data from Actions, you'll use xrGetActionState calls: there's one for each type of Action. Pose actions are a little bit different: they continue updating all the time, not just at xrSyncActions time, because tracking is latency and time sensitive. To get the data from a pose Action you'll usually just create an XrSpace for it, and then use xrLocateSpace. Only the active/inactive state of a pose Action is controlled by xrSyncActions.

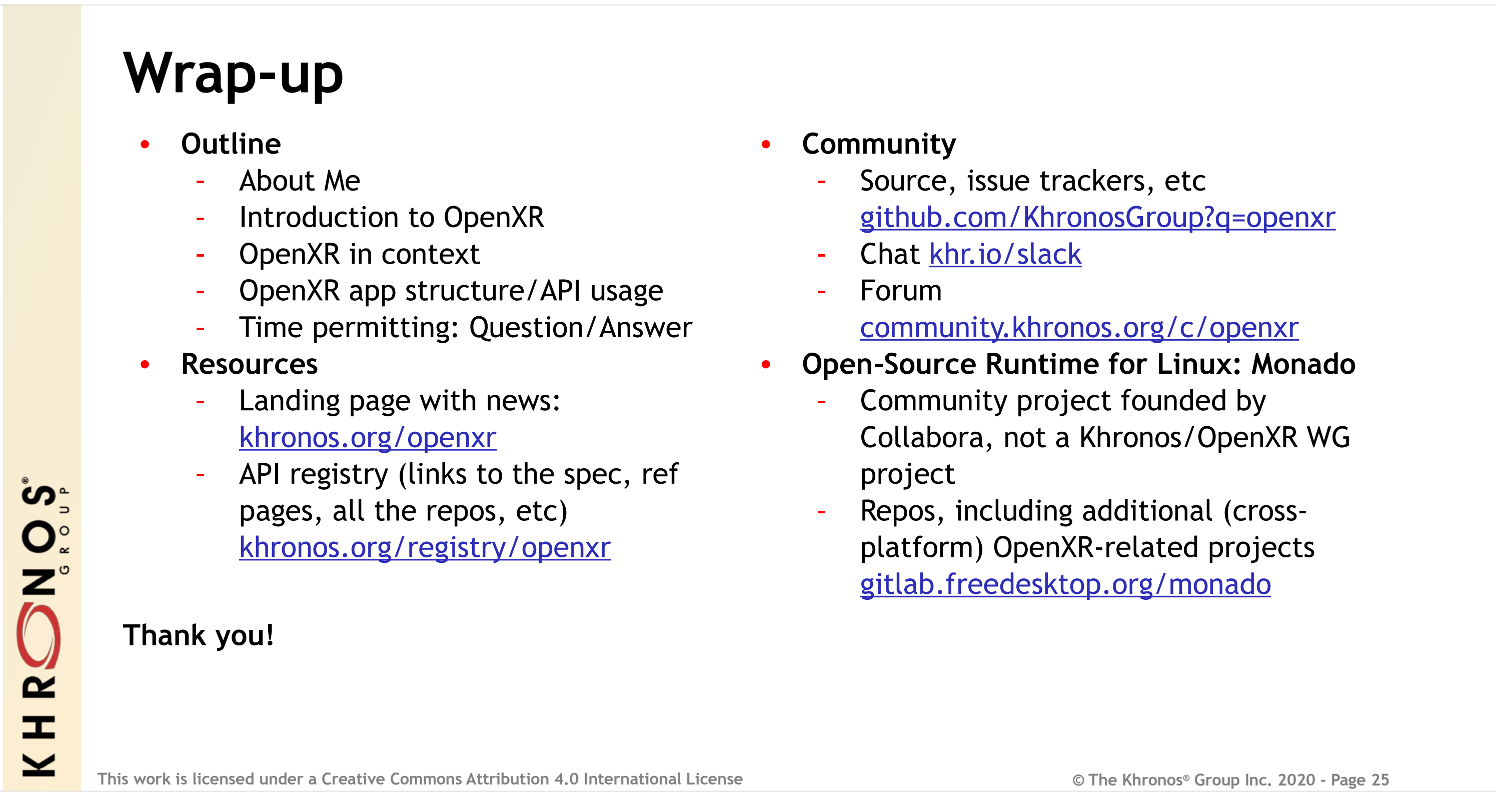
Typically you'll process most of your input either before you call xrWaitFrame and xrBeginFrame, or after xrEndFrame.

Events

- xrPollEvents
  - Requires an Instance
  - Many events only happen during a Session
- **Describes changes to**
  - Active interaction profile
  - Continuity of reference spaces/tracking
  - Session state
- **Provide an XrEventDataBuffer for the runtime to populate with an event of some other type**



There is a per instance event queue that contains a range of events. This queue must be polled on a regular basis. Typically polling once a simulation frame is a good idea. xrPollEvents requires an instance, however many events only happen during a Session. These events can describe changes to the active interaction profile, continuity of reference spaces and tracking, changes in the Session state, and other things. You provide an XrEventDataBuffer to xrPollEvents and the runtime populates it with an event of some other type. You have to make sure you set the type field to XR_TYPE_EVENT_DATA_BUFFER before the call, and then when you get it back from xrPollEvents, check that type value and reinterpret that structure accordingly.



Wrap-up

- **Outline**
  - About Me
  - Introduction to OpenXR
  - OpenXR in context
  - OpenXR app structure/API usage
  - Time permitting: Question/Answer
- **Resources**
  - Landing page with news: khronos.org/openxr
  - API registry (links to the spec, ref pages, all the repos, etc) khronos.org/registry/openxr
- **Community**
  - Source, issue trackers, etc github.com/KhronosGroup?q=openxr
  - Chat khr.io/slack
  - Forum community.khronos.org/c/openxr
- **Open-Source Runtime for Linux: Monado**
  - Community project founded by Collabora, not a Khronos/OpenXR WG project
  - Repos, including additional (cross-platform) OpenXR-related projects gitlab.freedesktop.org/monado

Thank you!

Thanks for your time! Hopefully you found this introduction to OpenXR and exploration of the structure of an OpenXR application to be helpful. I've put a number of resources on this slide that you can follow for additional information. If you have questions that we can't get to during this session, please feel free to drop by one of the community locations for the OpenXR group and leave a note there with your question. I or someone else in the community will be happy to respond to you.

OpenXR Master Class

Ryan A. Pavlik, Ph.D.
Principal Software Engineer, Collabora, Ltd.
OpenXR Working Group Spec. Editor