The OpenXR Specification

Version 1.0.16, Tue, 11 May 2021 20:12:12 +0000: from git ref release-1.0.16 commit:
862c84a9dcd227450bdc2f8de304567c2ba8722d

Table of Contents

1. Introduction . . . 2
   1.1. What is OpenXR? . . . 2
   1.2. The Programmer's View of OpenXR . . . 2
   1.3. The Implementor's View of OpenXR . . . 2
   1.4. Our View of OpenXR . . . 3
   1.5. Filing Bug Reports . . . 3
   1.6. Document Conventions . . . 3
2. Fundamentals . . . 5
   2.1. API Version Numbers and Semantics . . . 5
   2.2. String Encoding . . . 7
   2.3. Threading Behavior . . . 7
   2.4. Multiprocessing Behavior . . . 8
   2.5. Runtime . . . 8
   2.6. Extensions . . . 8
   2.7. API Layers . . . 9
   2.8. Return Codes . . . 16
   2.9. Handles . . . 23
   2.10. Object Handle Types . . . 23
   2.11. Buffer Size Parameters . . . 24
   2.12. Time . . . 26
   2.13. Duration . . . 27
   2.14. Prediction Time Limits . . . 28
   2.15. Colors . . . 28
   2.16. Coordinate System . . . 29
   2.17. Common Object Types . . . 32
   2.18. Angles . . . 35
   2.19. Boolean Values . . . 36
   2.20. Events . . . 36
   2.21. System resource lifetime . . . 41
3. API Initialization . . . 42
   3.1. Exported Functions . . . 42
   3.2. Function Pointers . . . 42
4. Instance . . . 46
   4.1. API Layers and Extensions . . . 46
   4.2. Instance Lifecycle . . . 52
   4.3. Instance Information . . . 57

4.4. Platform-Specific Instance Creation . . . 59
4.5. Instance Enumerated Type String Functions . . . 60
5. System . . . 63
5.1. Form Factors . . . 63
5.2. Getting the XrSystemId . . . 64
5.3. System Properties . . . 67
6. Path Tree and Semantic Paths . . . 71
6.1. Path Atom Type . . . 71
6.2. Well-Formed Path Strings . . . 73
6.3. Reserved Paths . . . 76
6.4. Interaction Profile Paths . . . 81
7. Spaces . . . 88
7.1. Reference Spaces . . . 89
7.2. Action Spaces . . . 94
7.3. Space Lifecycle . . . 95
7.4. Locating Spaces . . . 102
8. View Configurations . . . 110
8.1. Primary View Configurations . . . 110
8.2. View Configuration API . . . 111
8.3. Example View Configuration Code . . . 118
9. Session . . . 121
9.1. Session Lifecycle . . . 122
9.2. Session Control . . . 125
9.3. Session States . . . 130
10. Rendering . . . 135
10.1. Swapchain Image Management . . . 135
10.2. View and Projection State . . . 152
10.3. Frame Synchronization . . . 157
10.4. Frame Submission . . . 161
11. Input and Haptics . . . 179
11.1. Action Overview . . . 179
11.2. Action Sets . . . 181
11.3. Creating Actions . . . 185
11.4. Suggested Bindings . . . 190
11.5. Reading Input Action State . . . 200
11.6. Output Actions and Haptics . . . 211
11.7. Input Action State Synchronization . . . 216
11.8. Action Sources . . . 219

12. List of Extensions 225
12.1. XR_KHR_android_create_instance 226
12.2. XR_KHR_android_surface_swapchain 229
12.3. XR_KHR_android_thread_settings 232
12.4. XR_KHR_binding_modification 235
12.5. XR_KHR_composition_layer_color_scale_bias 238
12.6. XR_KHR_composition_layer_cube 241
12.7. XR_KHR_composition_layer_cylinder 244
12.8. XR_KHR_composition_layer_depth 248
12.9. XR_KHR_composition_layer_equirect 251
12.10. XR_KHR_composition_layer_equirect2 254
12.11. XR_KHR_convert_timespec_time 257
12.12. XR_KHR_D3D11_enable 261
12.13. XR_KHR_D3D12_enable 266
12.14. XR_KHR_loader_init 273
12.15. XR_KHR_loader_init_android 275
12.16. XR_KHR_opengl_enable 277
12.17. XR_KHR_opengl_es_enable 288
12.18. XR_KHR_visibility_mask 294
12.19. XR_KHR_vulkan_enable 300
12.20. XR_KHR_vulkan_enable2 313
12.21. XR_KHR_vulkan_swapchain_format_list 328
12.22. XR_KHR_win32_convert_performance_counter_time 331
12.23. XR_EXT_conformance_automation 334
12.24. XR_EXT_debug_utils 342
12.25. XR_EXT_eye_gaze_interaction 367
12.26. XR_EXT_hand_joints_motion_range 373
12.27. XR_EXT_hand_tracking 376
12.28. XR_EXT_hp_mixed_reality_controller 395
12.29. XR_EXT_performance_settings 396
12.30. XR_EXT_samsung_odyssey_controller 406
12.31. XR_EXT_thermal_query 408
12.32. XR_EXT_view_configuration_depth_range 411
12.33. XR_EXT_win32_appcontainer_compatible 414
12.34. XR_EPIC_view_configuration_fov 415
12.35. XR_FB_android_surface_swapchain_create 417
12.36. XR_FB_color_space 420
12.37. XR_FB_display_refresh_rate 427

12.38. XR_FB_swapchain_update_state 433
12.39. XR_HTC_vive_cosmos_controller_interaction 439
12.40. XR_HUAWEI_controller_interaction 441
12.41. XR_MND_headless 443
12.42. XR_MND_swapchain_usage_input_attachment_bit 445
12.43. XR_MSFT_controller_model 446
12.44. XR_MSFT_first_person_observer 459
12.45. XR_MSFT_hand_interaction 461
12.46. XR_MSFT_hand_tracking_mesh 463
12.47. XR_MSFT_holographic_window_attachment 482
12.48. XR_MSFT_perception_anchor_interop 486
12.49. XR_MSFT_secondary_view_configuration 490
12.50. XR_MSFT_spatial_anchor 502
12.51. XR_MSFT_spatial_graph_bridge 509
12.52. XR_MSFT_unbounded_reference_space 513
12.53. XR_OCULUS_android_session_state_enable 515
12.54. XR_VALVE_analog_threshold 517
12.55. XR_VARJO_composition_layer_depth_test 519
12.56. XR_VARJO_environment_depth_estimation 522
12.57. XR_VARJO_foveated_rendering 525
12.58. XR_VARJO_quad_views 531
13. List of Provisional Extensions 533
13.1. XR_EXTX_overlay 533
13.2. XR_MNDX_egl_enable 540
Appendix 543
Code Style Conventions 543
Application Binary Interface 543
Glossary 553
Abbreviations 554
Dedication (Informative) 556
Contributors (Informative) 558
Index 561

Chapter 1. Introduction

This chapter is informative except for the section on Normative Terminology.

This document, referred to as the "OpenXR Specification" or just the "Specification" hereafter, describes OpenXR: what it is, how it acts, and what is required to implement it. We assume that the reader has a basic understanding of computer graphics and the technologies involved in virtual and augmented reality. This means familiarity with the essentials of computer graphics algorithms and terminology, modern GPUs (Graphic Processing Units), tracking technologies, head mounted devices, and input modalities.

The canonical version of the Specification is available in the official OpenXR Registry, located at URL

http://www.khronos.org/registry/openxr/

1.1. What is OpenXR?

OpenXR is an API (Application Programming Interface) for XR applications. XR refers to a continuum of real-and-virtual combined environments generated by computers through human-machine interaction and is inclusive of the technologies associated with virtual reality (VR), augmented reality (AR) and mixed reality (MR). OpenXR is the interface between an application and an in-process or out-of-process "XR runtime system", or just "runtime" hereafter. The runtime may handle such functionality as frame composition, peripheral management, and raw tracking information.

Optionally, a runtime may support device layer plugins which allow access to a variety of hardware across a commonly defined interface.

1.2. The Programmer's View of OpenXR

To the application programmer, OpenXR is a set of functions that interface with a runtime to perform commonly required operations such as accessing controller/peripheral state, getting current and/or predicted tracking positions, and submitting rendered frames.

A typical OpenXR program begins with a call to create an instance which establishes a connection to a runtime. Then a call is made to create a system which selects for use a physical display and a subset of input, tracking, and graphics devices. Subsequently a call is made to create buffers into which the application will render one or more views using the appropriate graphics APIs for the platform. Finally calls are made to create a session and begin the application's XR rendering loop.

1.3. The Implementor's View of OpenXR

To the runtime implementor, OpenXR is a set of functions that control the operation of the XR system and establishes the lifecycle of a XR application.

The implementor's task is to provide a software library on the host which implements the OpenXR API, while mapping the work for each OpenXR function to the graphics hardware as appropriate for the capabilities of the device.

1.4. Our View of OpenXR

We view OpenXR as a mechanism for interacting with VR/AR/MR systems in a platform-agnostic way.

We expect this model to result in a specification that satisfies the needs of both programmers and runtime implementors. It does not, however, necessarily provide a model for implementation. A runtime implementation **must** produce results conforming to those produced by the specified methods, but **may** carry out particular procedures in ways that are more efficient than the one specified.

1.5. Filing Bug Reports

Issues with and bug reports on the OpenXR Specification and the API Registry **can** be filed in the Khronos OpenXR GitHub repository, located at URL

https://github.com/KhronosGroup/OpenXR-Docs

Please tag issues with appropriate labels, such as "Specification", "Ref Pages" or "Registry", to help us triage and assign them appropriately. Unfortunately, GitHub does not currently let users who do not have write access to the repository set GitHub labels on issues. In the meantime, they **can** be added to the title line of the issue set in brackets, e.g. "[Specification]".

1.6. Document Conventions

The OpenXR specification is intended for use by both implementors of the API and application developers seeking to make use of the API, forming a contract between these parties. Specification text may address either party; typically the intended audience can be inferred from context, though some sections are defined to address only one of these parties. (For example, Valid Usage sections only address application developers). Any requirements, prohibitions, recommendations or options defined by normative terminology are imposed only on the audience of that text.

1.6.1. Normative Terminology

The key words **must**, **required**, **should**, **may**, and **optional** in this document, when denoted as above, are to be interpreted as described in RFC 2119:

https://tools.ietf.org/html/rfc2119

must

When used alone, this word, or the term **required**, means that the definition is an absolute requirement of the specification. When followed by **not** ("**must** not"), the phrase means that the definition is an absolute prohibition of the specification.

should

When used alone, this word means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course. When followed by **not** (“**should** not”), the phrase means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications **should** be understood and the case carefully weighed before implementing any behavior described with this label.

may

This word, or the adjective **optional**, means that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item.

The additional terms **can** and **cannot** are to be interpreted as follows:

can

This word means that the particular behavior described is a valid choice for an application, and is never used to refer to runtime behavior.

cannot

This word means that the particular behavior described is not achievable by an application, for example, an entry point does not exist.

> There is an important distinction between **cannot** and **must not**, as used in this Specification. **Cannot** means something the application literally is unable to express or accomplish through the API, while **must not** means something that the application is capable of expressing through the API, but that the consequences of doing so are undefined and potentially unrecoverable for the runtime.

Chapter 2. Fundamentals

2.1. API Version Numbers and Semantics

Multi-part version numbers are used in several places in the OpenXR API.

```
typedef uint64_t XrVersion;
```

In each such use, the API major version number, minor version number, and patch version number are packed into a 64-bit integer, referred to as `XrVersion`, as follows:

Version Numbers

- The major version number is a 16-bit integer packed into bits 63-48.
- The minor version number is a 16-bit integer packed into bits 47-32.
- The patch version number is a 32-bit integer packed into bits 31-0.

Differences in any of the version numbers indicate a change to the API, with each part of the version number indicating a different scope of change, as follows.

> *Note*
>
> The rules below apply to OpenXR versions 1.0 or later. Prerelease versions of OpenXR may use different rules for versioning.

A difference in patch version numbers indicates that some usually small part of the specification or header has been modified, typically to fix a bug, and **may** have an impact on the behavior of existing functionality. Differences in the patch version number **must** affect neither full compatibility nor backwards compatibility between two versions, nor **may** it add additional interfaces to the API.

A difference in minor version numbers indicates that some amount of new functionality has been added. This will usually include new interfaces in the header, and **may** also include behavior changes and bug fixes. Functionality **may** be deprecated in a minor revision, but **must** not be removed. When a new minor version is introduced, the patch version is reset to 0, and each minor revision maintains its own set of patch versions. Differences in the minor version number **should** not affect backwards compatibility, but will affect full compatibility.

A difference in major version numbers indicates a large set of changes to the API, potentially including new functionality and header interfaces, behavioral changes, removal of deprecated features, modification or outright replacement of any feature, and is thus very likely to break compatibility.

Differences in the major version number will typically require significant modification to application code in order for it to function properly.

The following table attempts to detail the changes that may occur versus when they must not be updated (indicating the next version number must be updated instead) during an update to any of the major, minor, or patch version numbers:

Table 1. Scenarios Which May Cause a Version Change

Reason	***Major Version***	***Minor Version***	***Patch Version***
*Extensions Added/Removed**	**may**	**may**	**may**
*Spec-Optional Behavior Changed**	**may**	**may**	**may**
*Spec Required Behavior Changed**	**may**	**may**	**must** not
*Core Interfaces Added**	**may**	**may**	**must** not
*Weak Deprecation**	**may**	**may**	**must** not
*Strong Deprecation**	**may**	**must** not	**must** not
*Core Interfaces Changed/Removed**	**may**	**must** not	**must** not

In the above table, the following identify the various cases in detail:

Extensions Added/Removed	An extension may be added or removed with a change at this patch level.
Specification-Optional Behavior Changed	Some optional behavior laid out in this specification has changed. Usually this will involve a change in behavior that is marked with the normatives **should** or **may**. For example, a runtime that previously did not validate a particular use case may now begin validating that use case.
Specification-Required Behavior Changed	A behavior of runtimes that is required by this specification **may** have changed. For example, a previously optional validation **may** now have become mandatory for runtimes.
Core Interfaces Added	New interfaces **may** have been added to this specification (and to the OpenXR header file) in revisions at this level.

Weak Deprecation	An interface **may** have been weakly deprecated at this level. This **may** happen if there is now a better way to accomplish the same thing. Applications making this call should behave the same as before the deprecation, but following the new path may be more performant, less latent, or otherwise yield better results. It is possible that some runtimes **may** choose to give run-time warnings that the feature has been weakly deprecated and will likely be strongly deprecated or removed in the future.
Strong Deprecation	An interface **may** have been strongly deprecated at this level. This means that the interface **must** still exist (so applications that are compiled against it will still run) but it **may** now be a no-op, or it **may** be that its behavior has been significantly changed. It **may** be that this functionality is no longer necessary, or that its functionality has been subsumed by another call. This should not break an application, but some behavior **may** be different or unanticipated.
Interfaces Changed/Removed	An interface **may** have been changed — with different parameters or return types — at this level. An interface or feature **may** also have been removed entirely. It is almost certain that rebuilding applications will be required.

2.2. String Encoding

This API uses strings as input and output for some functions. Unless otherwise specified, all such strings are `NULL` terminated UTF-8 encoded case-sensitive character arrays.

2.3. Threading Behavior

The OpenXR API is intended to provide scalable performance when used on multiple host threads. All functions **must** support being called concurrently from multiple threads, but certain parameters, or components of parameters are defined to be externally synchronized. This means that the caller **must** guarantee that no more than one thread is using such a parameter at a given time.

More precisely, functions use simple stores to update software structures representing objects. A parameter declared as externally synchronized **may** have its software structures updated at any time during the host execution of the function. If two functions operate on the same object and at least one of the functions declares the object to be externally synchronized, then the caller **must** guarantee not only that the functions do not execute simultaneously, but also that the two functions are separated by an appropriate memory barrier if needed.

For all functions which destroy an object handle, the application **must** externally synchronize the object handle parameter and any child handles.

Externally Synchronized Parameters

- The `instance` parameter, and any child handles, in xrDestroyInstance
- The `session` parameter, and any child handles, in xrDestroySession
- The `space` parameter, and any child handles, in xrDestroySpace
- The `swapchain` parameter, and any child handles, in xrDestroySwapchain
- The `actionSet` parameter, and any child handles, in xrDestroyActionSet
- The `action` parameter, and any child handles, in xrDestroyAction
- The `objectHandle` member of the `nameInfo` parameter in xrSetDebugUtilsObjectNameEXT
- The `instance` parameter, and any child handles, in xrCreateDebugUtilsMessengerEXT
- The `messenger` parameter in xrDestroyDebugUtilsMessengerEXT
- The `handTracker` parameter, and any child handles, in xrDestroyHandTrackerEXT

Implicit Externally Synchronized Parameters

- The `session` parameter by any other xrWaitFrame call in xrWaitFrame
- The XrInstance used to create `messenger`, and all of its child handles in xrDestroyDebugUtilsMessengerEXT

2.4. Multiprocessing Behavior

The OpenXR API does not explicitly recognize nor require support for multiple processes using the runtime simultaneously, nor does it prevent a runtime from providing such support.

2.5. Runtime

An OpenXR runtime is software which implements the OpenXR API. There **may** be more than one OpenXR runtime installed on a system, but only one runtime can be active at any given time.

2.6. Extensions

OpenXR is an extensible API that **can** grow through the addition of new features. Similar to other Khronos APIs, extensions **can** be used to expose new OpenXR functions or modify the behavior of existing OpenXR functions. Extensions are optional and therefore **must** be enabled by the application before the extended functionality is made available. Because extensions are optional, they **may** be implemented only on a subset of runtimes, graphics platforms, or operating systems. Therefore, an application **should** first query which extensions are available before enabling.

The application queries the available list of extensions using the xrEnumerateInstanceExtensionProperties function. Once an application determines which target extensions are supported, it **can** enable some subset of them during the call to xrCreateInstance.

OpenXR extensions have unique names that convey information about what functionality is provided. The names have the following format:

Extension Name Formatting

- The prefix "`XR_`" to identify this as an OpenXR extension
- A string identifier for the vendor tag, which corresponds to the company or group exposing the extension. The vendor tag **must** use only uppercase letters and decimal digits. Some examples include:
  - "`KHR`" for Khronos extensions, supported by multiple vendors.
  - "`EXT`" for non-Khronos extensions supported by multiple vendors.
- An underscore "`_`".
- A string uniquely identifying the extension. The string is a compound of substrings which **must** use only lower case letters and decimal digits. The substrings are delimited with single underscores.

For example: `XR_KHR_composition_layer_cube` is an OpenXR extension created by the Khronos (KHR) OpenXR Working Group to support cube composition layers.

The public list of available extensions known at the time of this specification being generated appears in the List of Extensions appendix at the end of this document.

2.7. API Layers

OpenXR is designed to be a layered API, which means that a user or application **may** insert API layers between the application and the runtime implementation. These API layers provide additional functionality by intercepting OpenXR functions from the layer above and then performing different operations than would otherwise be performed without the layer. In the simplest cases, the layer simply calls the next layer down with the same arguments, but a more complex layer may implement API functionality that is not present in the layers or runtime below it. This mechanism is essentially an architected "function shimming" or "intercept" feature that is designed into OpenXR and meant to replace more informal methods of "hooking" API calls.

2.7.1. Examples of API Layers

Validation Layer

The layered API approach employed by OpenXR allows for the expensive validation of correct API usage to be implemented in a "validation" layer. This layer allows the application developer to develop their application with the validation layer active to ensure that the application is using the API correctly. The validation layer confirms that the application has set up object state correctly, has provided the required data for each function, ensures that required resources are available, etc. If the validation layer detects a problem, it issues an error message that can be logged or captured by the application via a callback. After the developer has determined that the application is correct, they turn off the validation layer to allow the application to run in a production environment without repeatedly incurring the validation expense.

API Logging Layer

Another example of an API layer is an API logging layer that simply serializes all the API calls to an output sink in a text format, including printing out argument values and structure contents.

API Trace Layer

A related API trace layer produces a trace file that contains all the information provided to the API so that the trace file can be played back by a replay program.

2.7.2. Naming API Layers

To organize API layer names and prevent collisions in the API layer name namespace, API layers **must** be named using the following convention:

```
XR_APILAYER_<VENDOR-TAG>_short_name
```

Vendors are responsible for registering a vendor tag with the OpenXR working group and just like for implementors, they must maintain their vendor namespace.

Example of an API layer name produced by the Acme company for the "check best practices" API layer:

```
XR_APILAYER_ACME_check_best_practices
```

2.7.3. Activating API Layers

Application Activation

Applications **can** determine the API layers that are available to them by calling the xrEnumerateApiLayerProperties function to obtain a list of available API layers. Applications then **can** select the desired API layers from this list and provide them to the xrCreateInstance function when creating an instance.

System Activation

Application users or users performing roles such as system integrator or system administrator **may** configure a system to activate API layers without involvement from the applications. These platform-dependent steps **may** include the installation of API layer-related files, setting environment variables, or other platform-specific operations. The options that are available for configuring the API layers in this manner are also dependent on the platform and/or runtime.

2.7.4. API Layer Extensions

API layers **may** implement OpenXR functions that may or may not be supported by the underlying runtime. In order to expose these new features, the API layer must expose this functionality in the form of an OpenXR extension. It **must** not expose new OpenXR functions without an associated extension.

For example, an OpenXR API-logging API layer might expose an API function to allow the application to turn logging on for only a portion of its execution. Since new functions **must** be exposed through an extension, the vendor has created an extension called `XR_ACME_logging_on_off` to contain these new functions. The application **should** query if the API layer supports the extension and then, only if it exists, enable both the extension and the API layer by name during xrCreateInstance.

To find out what extensions an API layer supports, an application **must** first verify that the API layer exists on the current system by calling xrEnumerateApiLayerProperties. After verifying an API layer of interest exists, the application then **should** call xrEnumerateInstanceExtensionProperties and provide the API layer name as the first parameter. This will return the list of extensions implemented internally in that API layer.

2.7.5. Type Aliasing

Type aliasing refers to the situation in which the actual type of a element does not match the declared type. Some C and C++ compilers can be configured to assume that the actual type matches the declared type, and may be so configured by default at common optimization levels. Without this, otherwise undefined behavior may occur. This compiler feature is typically referred to as "strict aliasing," and it can usually be enabled or disabled via compiler options. The OpenXR specification does not support strict aliasing, as there are some cases in which an application intentionally provides a struct with a type that differs from the declared type. For example, XrFrameEndInfo::`layers` is an array of type `const` XrCompositionLayerBaseHeader `* const`. However, the array **must** be of one of the specific layer types, such as XrCompositionLayerQuad. Similarly, xrEnumerateSwapchainImages accepts an array of XrSwapchainImageBaseHeader, whereas the actual type passed **must** be an array of a type such as XrSwapchainImageVulkanKHR. For OpenXR to work correctly, the compiler **must** support the type aliasing described here.

```
#if !defined(XR_MAY_ALIAS)
#if defined(__clang__) || (defined(__GNUC__) && (__GNUC__ > 4))
#define XR_MAY_ALIAS __attribute__((__may_alias__))
#else
#define XR_MAY_ALIAS
#endif
#endif
```

As a convenience, some types and pointers that are known at specification time to alias values of different types have been annotated with the XR_MAY_ALIAS definition. If this macro is not defined before including OpenXR headers, and a new enough Clang or GCC compiler is used, it will be defined to the compiler-specific attribute annotation to inform these compilers that those pointers may alias. However, there is no guarantee that all aliasing types or pointers have been correctly marked with this macro, so thorough testing is still recommended if you choose (at your own risk) to permit your compiler to perform type-based aliasing analysis.

2.7.6. Valid Usage

Valid usage defines a set of conditions which **must** be met in order to achieve well-defined run-time behavior in an application. These conditions depend only on API state, and the parameters or objects whose usage is constrained by the condition.

Some valid usage conditions have dependencies on runtime limits or feature availability. It is possible to validate these conditions against the API's minimum or maximum supported values for these limits and features, or some subset of other known values.

Valid usage conditions **should** apply to a function or structure where complete information about the condition would be known during execution of an application. This is such that a validation API layer or linter **can** be written directly against these statements at the point they are specified.

2.7.7. Implicit Valid Usage

Some valid usage conditions apply to all functions and structures in the API, unless explicitly denoted otherwise for a specific function or structure. These conditions are considered implicit. Implicit valid usage conditions are described in detail below.

Valid Usage for Object Handles

Any input parameter to a function that is an object handle **must** be a valid object handle, unless otherwise specified. An object handle is valid if and only if:

Object Handle Validity Conditions

- it has been created or allocated by a previous, successful call to the API,
- it has not been destroyed by a previous call to the API, and
- its parent handle is also valid.

There are contexts in which an object handle is **optional** or otherwise unspecified. In those cases, the API uses XR_NULL_HANDLE, which has the integer value `0`.

Valid Usage for Pointers

Any parameter that is a pointer **must** be a valid pointer when the specification indicates that the runtime uses the pointer. A pointer is valid if and only if it points at memory containing values of the number and type(s) expected by the function, and all fundamental types accessed through the pointer (e.g. as elements of an array or as members of a structure) satisfy the alignment requirements of the host processor.

Valid Usage for Enumerated Types

Any parameter of an enumerated type **must** be a valid enumerant for that type. An enumerant is valid if and only if the enumerant is defined as part of the enumerated type in question.

Valid Usage for Flags

A collection of flags is represented by a bitmask using the type `XrFlags64`:

```
typedef uint64_t XrFlags64;
```

Bitmasks are passed to many functions and structures to compactly represent options and are stored in memory defined by the `XrFlags64` type. But the API does not use the `XrFlags64` type directly. Instead, a `Xr*Flags` type is used which is an alias of the `XrFlags64` type. The API also defines a set of constant bit definitions used to set the bitmasks.

Any `Xr*Flags` member or parameter used in the API **must** be a valid combination of bit flags. A valid combination is either zero or the bitwise `OR` of valid bit flags. A bit flag is valid if and only if:

Bit Flag Validity

- The bit flag is one of the constant bit definitions defined by the same `Xr*Flags` type as the `Xr*Flags` member or parameter. Valid flag values may also be defined by extensions.
- The flag is allowed in the context in which it is being used. For example, in some cases, certain bit flags or combinations of bit flags are mutually exclusive.

Valid Usage for Structure Types

Any parameter that is a structure containing a `type` member **must** have a value of `type` which is a valid XrStructureType value matching the type of the structure. As a general rule, the name of this value is obtained by taking the structure name, stripping the leading `Xr`, prefixing each capital letter with an underscore, converting the entire resulting string to upper case, and prefixing it with `XR_TYPE_`.

The only exceptions to this rule are API and Operating System names which are converted in a way that produces a more readable value:

Structure Type Format Exceptions

- OpenGL ⇒ _OPENGL
- OpenGLES ⇒ _OPENGL_ES
- EGL ⇒ _EGL
- D3D ⇒ _D3D
- VULKAN ⇒ _VULKAN

Valid Usage for Structure Pointer Chains

Any structure containing a `void* next` member **must** have a value of `next` that is either `NULL`, or points to a valid structure that also contains `type` and `next` member values. The set of structures connected by `next` pointers is referred to as a `next` chain.

In order to use a structure type defined by an extension in a `next` chain, the proper extension **must** have been previously enabled during xrCreateInstance. A runtime **must** ignore all unrecognized structures in a `next` chain, including those associated with an extension that has not been enabled.

Some structures for use in a chain are described in the core OpenXR specification and are mentioned in the Member Descriptions. Any structure described in this document intended for use in a chain is mentioned in a "See also" list in the implicit valid usage of the structure they chain to. Most chained structures are associated with extensions, and are described in the base OpenXR Specification under the List of Extensions. Vendor-specific extensions **may** be found there as well, or **may** only be available from the vendor's website or internal document repositories.

Unless otherwise specified: Chained structs which are output structs **may** be modified by the runtime with the exception of the type and next fields. Upon return from any function, all type and next fields in the chain **must** be unmodified.

Useful Base Structures

As a convenience to runtimes and layers needing to iterate through a structure pointer chain, the OpenXR API provides the following base structures:

The XrBaseInStructure structure is defined as:

```
typedef struct XrBaseInStructure {
    XrStructureType                    type;
    const struct XrBaseInStructure*    next;
} XrBaseInStructure;
```

Member Descriptions

- `type` is the XrStructureType of this structure. This base structure itself has no associated XrStructureType value.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

XrBaseInStructure can be used to facilitate iterating through a read-only structure pointer chain.

The XrBaseOutStructure structure is defined as:

```
typedef struct XrBaseOutStructure {
    XrStructureType               type;
    struct XrBaseOutStructure*    next;
} XrBaseOutStructure;
```

Member Descriptions

- `type` is the XrStructureType of this structure. This base structure itself has no associated XrStructureType value.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

XrBaseOutStructure can be used to facilitate iterating through a structure pointer chain that returns data back to the application.

These structures allow for some type safety and can be used by OpenXR API functions that operate on generic inputs and outputs.

Next Chain Structure Uniqueness

Applications **should** ensure that they create and insert no more than one occurrence of each type of extension structure in a given `next` chain. Other components of OpenXR (such as the OpenXR loader or an API Layer) **may** insert duplicate structures into this chain. This provides those components the ability to update a structure that appears in the `next` chain by making a modified copy of that same structure and placing the new version at the beginning of the chain. The benefit of allowing this duplication is each component is no longer required to create a copy of the entire `next` chain just to update one structure. When duplication is present, all other OpenXR components **must** process only the first instance of a structure of a given type, and then ignore all instances of a structure of that same type.

If a component makes such a structure copy, and the original structure is also used to return content, then that component **must** copy the necessary content from the copied structure and into the original version of the structure upon completion of the function prior to proceeding back up the call stack. This is to ensure that OpenXR behavior is consistent whether or not that particular OpenXR component is present and/or enabled on the system.

Valid Usage for Nested Structures

The above conditions also apply recursively to members of structures provided as input to a function, either as a direct argument to the function, or themselves a member of another structure.

Specifics on valid usage of each function are covered in their individual sections.

2.8. Return Codes

While the core API is not designed to capture incorrect usage, some circumstances still require return codes. Functions in the API return their status via return codes that are in one of the two categories below.

Return Code Categories

- Successful completion codes are returned when a function needs to communicate success or status information. All successful completion codes are non-negative values.
- Run time error codes are returned when a function needs to communicate a failure that could only be detected at run time. All run time error codes are negative values.

```
typedef enum XrResult {
    XR_SUCCESS = 0,
    XR_TIMEOUT_EXPIRED = 1,
    XR_SESSION_LOSS_PENDING = 3,
    XR_EVENT_UNAVAILABLE = 4,
    XR_SPACE_BOUNDS_UNAVAILABLE = 7,
    XR_SESSION_NOT_FOCUSED = 8,
    XR_FRAME_DISCARDED = 9,
    XR_ERROR_VALIDATION_FAILURE = -1,
    XR_ERROR_RUNTIME_FAILURE = -2,
    XR_ERROR_OUT_OF_MEMORY = -3,
    XR_ERROR_API_VERSION_UNSUPPORTED = -4,
    XR_ERROR_INITIALIZATION_FAILED = -6,
    XR_ERROR_FUNCTION_UNSUPPORTED = -7,
    XR_ERROR_FEATURE_UNSUPPORTED = -8,
    XR_ERROR_EXTENSION_NOT_PRESENT = -9,
    XR_ERROR_LIMIT_REACHED = -10,
    XR_ERROR_SIZE_INSUFFICIENT = -11,
    XR_ERROR_HANDLE_INVALID = -12,
    XR_ERROR_INSTANCE_LOST = -13,
    XR_ERROR_SESSION_RUNNING = -14,
    XR_ERROR_SESSION_NOT_RUNNING = -16,
    XR_ERROR_SESSION_LOST = -17,
    XR_ERROR_SYSTEM_INVALID = -18,
    XR_ERROR_PATH_INVALID = -19,
    XR_ERROR_PATH_COUNT_EXCEEDED = -20,
    XR_ERROR_PATH_FORMAT_INVALID = -21,
    XR_ERROR_PATH_UNSUPPORTED = -22,
    XR_ERROR_LAYER_INVALID = -23,
    XR_ERROR_LAYER_LIMIT_EXCEEDED = -24,
    XR_ERROR_SWAPCHAIN_RECT_INVALID = -25,
    XR_ERROR_SWAPCHAIN_FORMAT_UNSUPPORTED = -26,
    XR_ERROR_ACTION_TYPE_MISMATCH = -27,
    XR_ERROR_SESSION_NOT_READY = -28,
    XR_ERROR_SESSION_NOT_STOPPING = -29,
    XR_ERROR_TIME_INVALID = -30,
```

```
    XR_ERROR_REFERENCE_SPACE_UNSUPPORTED = -31,
    XR_ERROR_FILE_ACCESS_ERROR = -32,
    XR_ERROR_FILE_CONTENTS_INVALID = -33,
    XR_ERROR_FORM_FACTOR_UNSUPPORTED = -34,
    XR_ERROR_FORM_FACTOR_UNAVAILABLE = -35,
    XR_ERROR_API_LAYER_NOT_PRESENT = -36,
    XR_ERROR_CALL_ORDER_INVALID = -37,
    XR_ERROR_GRAPHICS_DEVICE_INVALID = -38,
    XR_ERROR_POSE_INVALID = -39,
    XR_ERROR_INDEX_OUT_OF_RANGE = -40,
    XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED = -41,
    XR_ERROR_ENVIRONMENT_BLEND_MODE_UNSUPPORTED = -42,
    XR_ERROR_NAME_DUPLICATED = -44,
    XR_ERROR_NAME_INVALID = -45,
    XR_ERROR_ACTIONSET_NOT_ATTACHED = -46,
    XR_ERROR_ACTIONSETS_ALREADY_ATTACHED = -47,
    XR_ERROR_LOCALIZED_NAME_DUPLICATED = -48,
    XR_ERROR_LOCALIZED_NAME_INVALID = -49,
    XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING = -50,
    XR_ERROR_RUNTIME_UNAVAILABLE = -51,
    XR_ERROR_ANDROID_THREAD_SETTINGS_ID_INVALID_KHR = -1000003000,
    XR_ERROR_ANDROID_THREAD_SETTINGS_FAILURE_KHR = -1000003001,
    XR_ERROR_CREATE_SPATIAL_ANCHOR_FAILED_MSFT = -1000039001,
    XR_ERROR_SECONDARY_VIEW_CONFIGURATION_TYPE_NOT_ENABLED_MSFT = -1000053000,
    XR_ERROR_CONTROLLER_MODEL_KEY_INVALID_MSFT = -1000055000,
    XR_ERROR_DISPLAY_REFRESH_RATE_UNSUPPORTED_FB = -1000101000,
    XR_ERROR_COLOR_SPACE_UNSUPPORTED_FB = -1000108000,
    XR_RESULT_MAX_ENUM = 0x7FFFFFFF
} XrResult;
```

All return codes in the API are reported via XrResult return values.

Some common suffixes shared across many of the return codes are defined below:

- `_INVALID`: The specified handle, atom or value is formatted incorrectly, or the specified handle was never created or has been destroyed.
- `_UNSUPPORTED`: The specified handle, atom, enumerant or value is formatted correctly but cannot be used for the lifetime of this function's parent handle.
- `_UNAVAILABLE`: The specified handle, atom, enumerant or value is supported by this function's parent handle but not at this moment.

Success Codes

Enum	Description
`XR_SUCCESS`	Function successfully completed.

Enum	Description
XR_TIMEOUT_EXPIRED	The specified timeout time occurred before the operation could complete.
XR_SESSION_LOSS_PENDING	The session will be lost soon.
XR_EVENT_UNAVAILABLE	No event was available.
XR_SPACE_BOUNDS_UNAVAILABLE	The space's bounds are not known at the moment.
XR_SESSION_NOT_FOCUSED	The session is not in the focused state.
XR_FRAME_DISCARDED	A frame has been discarded from composition.

Error Codes

Enum	Description
XR_ERROR_VALIDATION_FAILURE	The function usage was invalid in some way.
XR_ERROR_RUNTIME_FAILURE	The runtime failed to handle the function in an unexpected way that is not covered by another error result.
XR_ERROR_OUT_OF_MEMORY	A memory allocation has failed.
XR_ERROR_API_VERSION_UNSUPPORTED	The runtime does not support the requested API version.
XR_ERROR_INITIALIZATION_FAILED	Initialization of object could not be completed.
XR_ERROR_FUNCTION_UNSUPPORTED	The requested function was not found or is otherwise unsupported.
XR_ERROR_FEATURE_UNSUPPORTED	The requested feature is not supported.
XR_ERROR_EXTENSION_NOT_PRESENT	A requested extension is not supported.
XR_ERROR_LIMIT_REACHED	The runtime supports no more of the requested resource.
XR_ERROR_SIZE_INSUFFICIENT	The supplied size was smaller than required.
XR_ERROR_HANDLE_INVALID	A supplied object handle was invalid.
XR_ERROR_INSTANCE_LOST	The XrInstance was lost or could not be found. It will need to be destroyed and optionally recreated.
XR_ERROR_SESSION_RUNNING	The session is already running.
XR_ERROR_SESSION_NOT_RUNNING	The session is not yet running.
XR_ERROR_SESSION_LOST	The XrSession was lost. It will need to be destroyed and optionally recreated.
XR_ERROR_SYSTEM_INVALID	The provided XrSystemId was invalid.

Enum	Description
XR_ERROR_PATH_INVALID	The provided XrPath was not valid.
XR_ERROR_PATH_COUNT_EXCEEDED	The maximum number of supported semantic paths has been reached.
XR_ERROR_PATH_FORMAT_INVALID	The semantic path character format is invalid.
XR_ERROR_PATH_UNSUPPORTED	The semantic path is unsupported.
XR_ERROR_LAYER_INVALID	The layer was NULL or otherwise invalid.
XR_ERROR_LAYER_LIMIT_EXCEEDED	The number of specified layers is greater than the supported number.
XR_ERROR_SWAPCHAIN_RECT_INVALID	The image rect was negatively sized or otherwise invalid.
XR_ERROR_SWAPCHAIN_FORMAT_UNSUPPORTED	The image format is not supported by the runtime or platform.
XR_ERROR_ACTION_TYPE_MISMATCH	The API used to retrieve an action's state does not match the action's type.
XR_ERROR_SESSION_NOT_READY	The session is not in the ready state.
XR_ERROR_SESSION_NOT_STOPPING	The session is not in the stopping state.
XR_ERROR_TIME_INVALID	The provided XrTime was zero, negative, or out of range.
XR_ERROR_REFERENCE_SPACE_UNSUPPORTED	The specified reference space is not supported by the runtime or system.
XR_ERROR_FILE_ACCESS_ERROR	The file could not be accessed.
XR_ERROR_FILE_CONTENTS_INVALID	The file's contents were invalid.
XR_ERROR_FORM_FACTOR_UNSUPPORTED	The specified form factor is not supported by the current runtime or platform.
XR_ERROR_FORM_FACTOR_UNAVAILABLE	The specified form factor is supported, but the device is currently not available, e.g. not plugged in or powered off.
XR_ERROR_API_LAYER_NOT_PRESENT	A requested API layer is not present or could not be loaded.
XR_ERROR_CALL_ORDER_INVALID	The call was made without having made a previously required call.
XR_ERROR_GRAPHICS_DEVICE_INVALID	The given graphics device is not in a valid state. The graphics device could be lost or initialized without meeting graphics requirements.

Enum	Description
XR_ERROR_POSE_INVALID	The supplied pose was invalid with respect to the requirements.
XR_ERROR_INDEX_OUT_OF_RANGE	The supplied index was outside the range of valid indices.
XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED	The specified view configuration type is not supported by the runtime or platform.
XR_ERROR_ENVIRONMENT_BLEND_MODE_UNSUPPORTED	The specified environment blend mode is not supported by the runtime or platform.
XR_ERROR_NAME_DUPLICATED	The name provided was a duplicate of an already-existing resource.
XR_ERROR_NAME_INVALID	The name provided was invalid.
XR_ERROR_ACTIONSET_NOT_ATTACHED	A referenced action set is not attached to the session.
XR_ERROR_ACTIONSETS_ALREADY_ATTACHED	The session already has attached action sets.
XR_ERROR_LOCALIZED_NAME_DUPLICATED	The localized name provided was a duplicate of an already-existing resource.
XR_ERROR_LOCALIZED_NAME_INVALID	The localized name provided was invalid.
XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING	The xrGetGraphicsRequirements* call was not made before calling xrCreateSession.
XR_ERROR_RUNTIME_UNAVAILABLE	The loader was unable to find or load a runtime.
XR_ERROR_ANDROID_THREAD_SETTINGS_ID_INVALID_KHR	xrSetAndroidApplicationThreadKHR failed as thread id is invalid. (Added by the XR_KHR_android_thread_settings extension)
XR_ERROR_ANDROID_THREAD_SETTINGS_FAILURE_KHR	xrSetAndroidApplicationThreadKHR failed setting the thread attributes/priority. (Added by the XR_KHR_android_thread_settings extension)
XR_ERROR_CREATE_SPATIAL_ANCHOR_FAILED_MSFT	Spatial anchor could not be created at that location. (Added by the XR_MSFT_spatial_anchor extension)
XR_ERROR_SECONDARY_VIEW_CONFIGURATION_TYPE_NOT_ENABLED_MSFT	The secondary view configuration was not enabled when creating the session. (Added by the XR_MSFT_secondary_view_configuration extension)
XR_ERROR_CONTROLLER_MODEL_KEY_INVALID_MSFT	The controller model key is invalid. (Added by the XR_MSFT_controller_model extension)

Enum	Description
XR_ERROR_DISPLAY_REFRESH_RATE_UNSUPPORTED_FB	The display refresh rate is not supported by the platform. (Added by the XR_FB_display_refresh_rate extension)
XR_ERROR_COLOR_SPACE_UNSUPPORTED_FB	The color space is not supported by the runtime. (Added by the XR_FB_color_space extension)

2.8.1. Convenience Macros

```
#define XR_SUCCEEDED(result) ((result) >= 0)
```

A convenience macro that can be used to test if a function succeeded. This may be a qualified success such as XR_FRAME_DISCARDED.

```
#define XR_FAILED(result) ((result) < 0)
```

A convenience macro that can be used to test if a function has failed in some way.

```
#define XR_UNQUALIFIED_SUCCESS(result) ((result) == 0)
```

A convenience macro that can be used to test a function's failure. The XR_UNQUALIFIED_SUCCESS macro is a convenience macro which **may** be used to compare an XrResult to 0 (XR_SUCCESS) exclusively.

2.8.2. Validation

Except as noted below or in individual API specifications, valid API usage **may** be required by the runtime. Runtimes **may** choose to validate some API usage and return an appropriate error code.

Application developers **should** use validation layers to catch and eliminate errors during development. Once validated, applications **should** not enable validation layers by default.

If a function returns a run time error, unless otherwise specified any output parameters will have undefined contents, except that if the output parameter is a structure with type and next fields, those fields will be unmodified. Any output structures chained from next will also have undefined contents, except that the type and next will be unmodified.

Unless otherwise specified, errors do not affect existing OpenXR objects. Objects that have already been successfully created **may** still be used by the application.

XrResult code returns may be added to a given function in future versions of the specification. Runtimes **must** return only XrResult codes from the set documented for the given application API version.

Runtimes **must** ensure that incorrect usage by an application does not affect the integrity of the operating system, the API implementation, or other API client applications in the system, and does not allow one application to access data belonging to another application.

2.9. Handles

Objects which are allocated by the runtime on behalf of applications are represented by handles. Handles are opaque identifiers for objects whose lifetime is controlled by applications via the create and destroy functions. Example handle types include XrInstance, XrSession, and XrSwapchain. Handles which have not been destroyed are unique for a given application process, but **may** be reused after being destroyed. Unless otherwise specified, a successful handle creation function call returns a new unique handle. Unless otherwise specified, handles are implicitly destroyed when their parent handle is destroyed. Applications **may** destroy handles explicitly before the parent handle is destroyed, and **should** do so if no longer needed, in order to conserve resources. Runtimes **may** detect XR_NULL_HANDLE and other invalid handles passed where a valid handle is required and return `XR_ERROR_HANDLE_INVALID`. However, runtimes are not required to do so unless otherwise specified, and so use of any invalid handle **may** result in undefined behavior. When a function has an optional handle parameter, XR_NULL_HANDLE **must** be used unless passing a valid handle.

All functions that take a handle parameter **may** return `XR_ERROR_HANDLE_INVALID`.

Handles form a hierarchy in which child handles fall under the validity and lifetime of parent handles. For example, to create an XrSwapchain handle, applications must call xrCreateSwapchain and pass an XrSession handle. Thus XrSwapchain is a child handle to XrSession.

2.10. Object Handle Types

The type of an object handle used in a function is usually determined by the specification of that function, as discussed in Valid Usage for Object Handles. However, some functions accept or return object handle parameters where the type of the object handle is unknown at execution time and is not specified in the description of the function itself. For these functions, the XrObjectType **may** be used to explicitly specify the type of a handle.

For example, an information-gathering or debugging mechanism implemented in a runtime extension or API layer extension **may** return a list of object handles that are generated by the mechanism's operation. The same mechanism **may** also return a parallel list of object handle types that allow the recipient of this information to easily determine the types of the handles.

In general, anywhere an object handle of more than one type can occur, the object handle type **may** be provided to indicate its type.

```
typedef enum XrObjectType {
    XR_OBJECT_TYPE_UNKNOWN = 0,
    XR_OBJECT_TYPE_INSTANCE = 1,
    XR_OBJECT_TYPE_SESSION = 2,
    XR_OBJECT_TYPE_SWAPCHAIN = 3,
    XR_OBJECT_TYPE_SPACE = 4,
    XR_OBJECT_TYPE_ACTION_SET = 5,
    XR_OBJECT_TYPE_ACTION = 6,
    XR_OBJECT_TYPE_DEBUG_UTILS_MESSENGER_EXT = 1000019000,
    XR_OBJECT_TYPE_SPATIAL_ANCHOR_MSFT = 1000039000,
    XR_OBJECT_TYPE_HAND_TRACKER_EXT = 1000051000,
    XR_OBJECT_TYPE_MAX_ENUM = 0x7FFFFFFF
} XrObjectType;
```

The XrObjectType enumeration defines values, each of which corresponds to a specific OpenXR handle type. These values **can** be used to associate debug information with a particular type of object through one or more extensions.

The following table defines XrObjectType and OpenXR Handle relationships:

XrObjectType	OpenXR Handle Type
`XR_OBJECT_TYPE_UNKNOWN`	Unknown/Undefined Handle
`XR_OBJECT_TYPE_INSTANCE`	XrInstance
`XR_OBJECT_TYPE_SESSION`	XrSession
`XR_OBJECT_TYPE_SWAPCHAIN`	XrSwapchain
`XR_OBJECT_TYPE_SPACE`	XrSpace
`XR_OBJECT_TYPE_ACTION_SET`	XrActionSet
`XR_OBJECT_TYPE_ACTION`	XrAction

2.11. Buffer Size Parameters

Functions with input/output buffer parameters take on either parameter form or struct form, looking like one of the following examples, with the element type being `float` in this case:

Parameter form:

```
XrResult xrFunction(uint32_t elementCapacityInput, uint32_t* elementCountOutput, float*
elements);
```

Struct form:

```
XrResult xrFunction(XrBuffer* buffer);

struct XrBuffer {
    uint32_t              elementCapacityInput;
    uint32_t              elementCountOutput;
    float*                elements;
};
```

A two-call idiom **may** be employed, first calling `xrFunction` (with a valid `elementCountOutput` pointer if in parameter form), but passing `NULL` as `elements` and `0` as `elementCapacityInput`, to retrieve the required buffer size as number of elements (number of floats in this example). After allocating a buffer at least as large as `elementCountOutput` (in a struct) or the value pointed to by `elementCountOutput` (as parameters), a pointer to the allocated buffer **should** be passed as `elements`, along with the buffer's length in `elementCapacityInput`, to a second call to `xrFunction` to perform the retrieval of the data. In case that `elements` is a struct with `type` and `next` fields, the application **must** set the `type` to the correct value as well as `next` either to `NULL` or a struct with extension related data in which `type` and `next` also need to be well defined.

In the following discussion, "set `elementCountOutput`" should be interpreted as "set the value pointed to by `elementCountOutput`" in parameter form and "set the value of `elementCountOutput`" in struct form. These functions have the below-listed behavior with respect to the buffer size parameters:

Buffer Size Parameter Behavior

- The element capacity and count arguments precede the array to which they refer, in argument order.
- `elementCapacityInput` specifies the capacity in number of elements of the buffer to be written, or `0` to indicate a request for the required buffer size.
- Independent of `elementCapacityInput` or `elements` parameters, the function sets `elementCountOutput`. `elementCountOutput` **must** be a valid pointer if the function uses parameter form.
- Where the `elementCapacityInput` is `0`, the function sets `elementCountOutput` to the required size in number of elements and **must** return `XR_SUCCESS`. `elements` is ignored.
- Where the `elementCapacityInput` is non-zero but less than required, the function sets `elementCountOutput` to the required capacity, and **must** return `XR_ERROR_SIZE_INSUFFICIENT`. The data in `elements` is undefined.
- Where the `elementCapacityInput` is non-zero and the function returned successfully, the function sets `elementCountOutput` to the count of the elements that have been written to `elements`.
- Upon a failure for reasons unrelated to the element array capacity, the contents of `elementCountOutput` and `elements` are undefined.
- In the case that the element array refers to a string (is of type `char*`), `elementCapacityInput` and `elementCountOutput` refer to the string `strlen` plus `1` for a `NULL` terminator.

Some functions fill multiple buffers in one call. For these functions, the `elementCapacityInput`, `elementCountOutput` and `elements` parameters or fields are repeated, once per buffer, with different prefixes. In that case, the semantics above still apply, with the additional behavior that if any `elementCapacityInput` parameter or field is set to 0 by the application, the runtime **must** treat all `elementCapacityInput` values as if they were set to 0. If any `elementCapacityInput` value is too small to fit all elements of the buffer, `XR_ERROR_SIZE_INSUFFICIENT` **must** be returned, and the data in all buffers is undefined.

2.12. Time

Time is represented by a 64-bit signed integer representing nanoseconds (`XrTime`). The passage of time **must** be monotonic and not real-time (i.e. wall clock time). Thus the time is always increasing at a constant rate and is unaffected by clock changes, time zones, daylight savings, etc.

2.12.1. XrTime

```
typedef int64_t XrTime;
```

`XrTime` is a base value type that represents time as a signed 64-bit integer, representing the monotonically-increasing count of nanoseconds that have elapsed since a runtime-chosen epoch. `XrTime` always represents the time elasped since that constant epoch, rather than a duration or a time point relative to some moving epoch such as vsync time, etc. Durations are instead represented by `XrDuration`.

A single runtime **must** use the same epoch for all simultaneous applications. Time **must** be represented the same regardless of multiple processors or threads present in the system.

The period precision of time reported by the runtime is runtime-dependent, and **may** change. One nanosecond is the finest possible period precision. A runtime **may**, for example, report time progression with only microsecond-level granularity.

Time **must** not be assumed to correspond to a system clock time.

Unless specified otherwise, zero or a negative value is not a valid `XrTime`, and related functions **must** return error `XR_ERROR_TIME_INVALID`. Applications **must** not initialize such `XrTime` fields to a zero value. Instead, applications **should** always assign `XrTime` fields to the meaningful point in time they are choosing to reason about, such as a frame's predicted display time, or an action's last change time.

The behavior of a runtime is undefined when time overflows beyond the maximum positive value that can be represented by an `XrTime`. Runtimes **should** choose an epoch that minimizes the chance of overflow. Runtimes **should** also choose an epoch that minimizes the chance of underflow below 0 for applications performing a reasonable amount of historical pose lookback. For example, if the runtime chooses an epoch relative to its startup time, it **should** push the epoch into the past by enough time to avoid applications performing reasonable pose lookback from reaching a negative `XrTime` value.

An application cannot assume that the system's clock and the runtime's clock will maintain a constant relationship across frames and **should** avoid storing such an offset, as this may cause time drift. Applications **should** instead always use time interop functions to convert a relevant time point across the system's clock and the runtime's clock using extensions, for example, XR_KHR_win32_convert_performance_counter_time or XR_KHR_convert_timespec_time.

2.13. Duration

Duration refers to an elapsed period of time, as opposed to an absolute timepoint.

2.13.1. XrDuration

```
typedef int64_t XrDuration;
```

The difference between two timepoints is a duration, and thus the difference between two `XrTime` values is an `XrDuration` value.

Functions that refer to durations use `XrDuration` as opposed to `XrTime`.

```
#define XR_NO_DURATION 0
```

For the case of timeout durations, XR_NO_DURATION **may** be used to indicate that the timeout is immediate.

```
#define XR_INFINITE_DURATION 0x7fffffffffffffffLL
```

XR_INFINITE_DURATION is a special value that **may** be used to indicate that the timeout never occurs. A timeout with a duration that refers to the past has the same effect as a timeout of XR_NO_DURATION.

2.14. Prediction Time Limits

Some functions involve prediction. For example, xrLocateViews accepts a display time for which to return the resulting data. Prediction times provided by applications may refer to time in the past or the future. Times in the past **may** be interpolated historical data. Runtimes have different practical limits with respect to how far forward or backward prediction times can be accurate. There is no prescribed forward limit the application can successfully request predictions for, though predictions may become less accurate as they get farther into the future. With respect to backward prediction, the application can pass a prediction time equivalent to the timestamp of the most recently received pose plus as much as `50` milliseconds in the past to retrieve accurate historical data. Requested times predating this time window, or requested times predating the earliest received pose, **may** result in a best effort data whose accuracy reduced or unspecified.

2.15. Colors

The XrColor4f structure is defined as:

```
typedef struct XrColor4f {
    float    r;
    float    g;
    float    b;
    float    a;
} XrColor4f;
```

Member Descriptions

- `r` is the red component of the color.
- `g` is the green component of the color.
- `b` is the blue component of the color.
- `a` is the alpha component of the color.

Unless otherwise specified, colors are encoded as linear (not with sRGB nor other gamma compression) values with individual components being in the range of 0.0 through 1.0, and without the RGB components being premultiplied by the alpha component.

If color encoding is specified as being premultiplied by the alpha component, the RGB components are set to zero if the alpha component is zero.

2.16. Coordinate System

This API uses a Cartesian right-handed coordinate system.

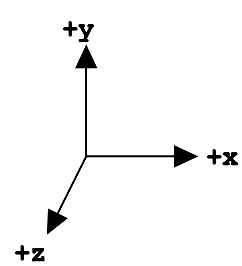


Figure 1. Right Handed Coordinate System

The conventions for mapping coordinate axes of any particular space to meaningful directions depend on and are documented with the description of the space.

The API uses 2D, 3D, and 4D floating-point vectors to describe points and directions in a space.

A two-dimensional vector is defined by the XrVector2f structure:

```
typedef struct XrVector2f {
    float    x;
    float    y;
} XrVector2f;
```

Member Descriptions

- `x` is the x coordinate of the vector.
- `y` is the y coordinate of the vector.

If used to represent physical distances (rather than e.g. normalized direction) and not otherwise specified, values **must** be in meters.

A three-dimensional vector is defined by the XrVector3f structure:

```
typedef struct XrVector3f {
    float    x;
    float    y;
    float    z;
} XrVector3f;
```

Member Descriptions

- `x` is the x coordinate of the vector.
- `y` is the y coordinate of the vector.
- `z` is the z coordinate of the vector.

If used to represent physical distances (rather than e.g. velocity or angular velocity) and not otherwise specified, values **must** be in meters.

A four-dimensional or homogeneous vector is defined by the XrVector4f structure:

```
typedef struct XrVector4f {
    float    x;
    float    y;
    float    z;
    float    w;
} XrVector4f;
```

Member Descriptions

- `x` is the x coordinate of the vector.
- `y` is the y coordinate of the vector.
- `z` is the z coordinate of the vector.
- `w` is the w coordinate of the vector.

If used to represent physical distances, `x`, `y`, and `z` values **must** be in meters.

Rotation is represented by a unit quaternion defined by the XrQuaternionf structure:

```
typedef struct XrQuaternionf {
    float    x;
    float    y;
    float    z;
    float    w;
} XrQuaternionf;
```

Member Descriptions

- `x` is the x coordinate of the quaternion.
- `y` is the y coordinate of the quaternion.
- `z` is the z coordinate of the quaternion.
- `w` is the w coordinate of the quaternion.

A pose is defined by the XrPosef structure:

```
typedef struct XrPosef {
    XrQuaternionf    orientation;
    XrVector3f       position;
} XrPosef;
```

Member Descriptions

- `orientation` is an XrQuaternionf representing the orientation within a space.
- `position` is an XrVector3f representing position within a space.

A construct representing a position and orientation within a space, with position expressed in meters, and orientation represented as a unit quaternion. When using XrPosef the rotation described by `orientation` is always applied before the translation described by `position`.

A runtime **must** return `XR_ERROR_POSE_INVALID` if the `orientation` norm deviates by more than 1% from unit length.

2.17. Common Object Types

Some types of OpenXR objects are used in multiple structures. Those include the `XrVector*f` and types specified above but also the following structures: offset, extents and rectangle.

Offsets are used to describe the magnitude of an offset in two dimensions.

A floating-point offset is defined by the structure:

```
typedef struct XrOffset2Df {
    float    x;
    float    y;
} XrOffset2Df;
```

Member Descriptions

- `x` the floating-point offset in the x direction.
- `y` the floating-point offset in the y direction.

This structure is used for component values that may be fractional (floating-point). If used to represent physical distances, values **must** be in meters.

An integer offset is defined by the structure:

```
typedef struct XrOffset2Di {
    int32_t    x;
    int32_t    y;
} XrOffset2Di;
```

Member Descriptions

- `x` the integer offset in the x direction.
- `y` the integer offset in the y direction.

This variant is for representing discrete values such as texels. For representing physical distances, the floating-point variant **must** be used instead.

Extents are used to describe the size of a rectangular region in two dimensions.

A two-dimensional floating-point extent is defined by the structure:

```
typedef struct XrExtent2Df {
    float    width;
    float    height;
} XrExtent2Df;
```

Member Descriptions

- `width` the floating-point width of the extent.
- `height` the floating-point height of the extent.

This structure is used for component values that may be fractional (floating-point). If used to represent physical distances, values **must** be in meters.

The `width` and `height` value **must** be non-negative.

A two-dimensional integer extent is defined by the structure:

```
typedef struct XrExtent2Di {
    int32_t    width;
    int32_t    height;
} XrExtent2Di;
```

Member Descriptions

- `width` the integer width of the extent.
- `height` the integer height of the extent.

This variant is for representing discrete values such as texels. For representing physical distances, the floating-point variant **must** be used instead.

The `width` and `height` value **must** be non-negative.

Rectangles are used to describe a specific rectangular region in two dimensions. Rectangles **must** include both an offset and an extent defined in the same units. For instance, if a rectangle is in meters, both offset and extent **must** be in meters.

A rectangle with floating-point values is defined by the structure:

```
typedef struct XrRect2Df {
    XrOffset2Df    offset;
    XrExtent2Df    extent;
} XrRect2Df;
```

Member Descriptions

- `offset` is the XrOffset2Df specifying the rectangle offset.
- `extent` is the XrExtent2Df specifying the rectangle extent.

This structure is used for component values that may be fractional (floating-point).

A rectangle with integer values is defined by the structure:

```c
typedef struct XrRect2Di {
    XrOffset2Di    offset;
    XrExtent2Di    extent;
} XrRect2Di;
```

Member Descriptions

- `offset` is the XrOffset2Di specifying the integer rectangle offset.
- `extent` is the XrExtent2Di specifying the integer rectangle extent.

This variant is for representing discrete values such as texels. For representing physical distances, the floating-point variant **must** be used instead.

2.18. Angles

Where a value is provided as a function parameter or as a structure member and will be interpreted as an angle, the value is defined to be in radians.

Field of view (FoV) is defined by the structure:

```c
typedef struct XrFovf {
    float    angleLeft;
    float    angleRight;
    float    angleUp;
    float    angleDown;
} XrFovf;
```

Member Descriptions

- `angleLeft` is the angle of the left side of the field of view. For a symmetric field of view this value is negative.
- `angleRight` is the angle of the right side of the field of view.
- `angleUp` is the angle of the top part of the field of view.
- `angleDown` is the angle of the bottom part of the field of view. For a symmetric field of view this value is negative.

Angles to the right of the center and upwards from the center are positive, and angles to the left of the center and down from the center are negative. The total horizontal field of view is `angleRight` minus `angleLeft`, and the total vertical field of view is `angleUp` minus `angleDown`. For a symmetric FoV, `angleRight` and `angleUp` will have positive values, `angleLeft` will be -`angleRight`, and `angleDown` will be -`angleUp`.

The angles **must** be specified in radians, and **must** be between -π/2 and π/2 exclusively.

When `angleLeft` > `angleRight`, the content of the view **must** be flipped horizontally. When `angleDown` > `angleUp`, the content of the view **must** be flipped vertically.

2.19. Boolean Values

```
typedef uint32_t XrBool32;
```

Boolean values used by OpenXR are of type `XrBool32` and are 32-bits wide as suggested by the name. The only valid values are the following:

Enumerant Descriptions

- `XR_TRUE` represents a true value.
- `XR_FALSE` represents a false value.

2.20. Events

Events are messages sent from the runtime to the application.

2.20.1. Event Polling

These events are placed in a queue and the application **must** read from the queue with regularity. Events are read from the queue one at a time via xrPollEvent. Every event is identified by an individual struct, with each struct beginning with an XrEventDataBaseHeader.

Example 1. Proper Method for Receiving OpenXR Event Data

```
XrInstance instance; // previously initialized

// Initialize an event buffer to hold the output.
XrEventDataBuffer event;
// Only the header needs to be initialized.
event.type = XR_TYPE_EVENT_DATA_BUFFER;
event.next = nullptr;
XrResult result = xrPollEvent(instance, &event);
if (result == XR_SUCCESS) {
    switch (event.type) {
        case XR_TYPE_EVENT_DATA_SESSION_STATE_CHANGED: {
            const XrEventDataSessionStateChanged& session_state_changed_event =
                *reinterpret_cast<XrEventDataSessionStateChanged*>(&event);
            // ...
            break;
        }
        case XR_TYPE_EVENT_DATA_INSTANCE_LOSS_PENDING: {
            const XrEventDataInstanceLossPending& instance_loss_pending_event =
                *reinterpret_cast<XrEventDataInstanceLossPending*>(&event);
            // ...
            break;
        }
    }
}
```

xrPollEvent

```
XrResult xrPollEvent(
    XrInstance                                  instance,
    XrEventDataBuffer*                          eventData);
```

xrPollEvent polls for the next event and returns an event if one is available. xrPollEvent returns immediately regardless of whether an event was available. The event (if present) is unilaterally removed from the queue if a valid XrInstance is provided. On return the `eventData` parameter is filled with the event's data and the type field is changed to the event's type. Runtimes **may** create valid next chains depending on enabled extensions, but they **must** guarantee that any such chains point only to objects which fit completely within the original XrEventDataBuffer pointed to by `eventData`.

Parameter Descriptions

- `instance` is a valid XrInstance.
- `eventData` is a pointer to a valid XrEventDataBuffer.

Valid Usage (Implicit)

- `instance` **must** be a valid XrInstance handle
- `eventData` **must** be a pointer to an XrEventDataBuffer structure

Return Codes

Success

- `XR_SUCCESS`
- `XR_EVENT_UNAVAILABLE`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_VALIDATION_FAILURE`

The runtime **must** discard queued events which contain destroyed or otherwise invalid handles.

Table 2. Event Descriptions

Event	Description
XrEventDataEventsLost	event queue has overflowed and some events were lost
XrEventDataInstanceLossPending	application is about to lose the instance
XrEventDataInteractionProfileChanged	active input form factor for one or more top level user paths has changed
XrEventDataReferenceSpaceChangePending	runtime will begin operating with updated space bounds
XrEventDataSessionStateChanged	application has changed lifecycle state

The XrEventDataBaseHeader structure is defined as:

```
typedef struct XrEventDataBaseHeader {
    XrStructureType    type;
    const void*        next;
} XrEventDataBaseHeader;
```

Parameter Descriptions

- `type` is the XrStructureType of this structure. This base structure itself has no associated XrStructureType value.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

The XrEventDataBaseHeader is a generic structure used to identify the common event data elements.

Upon receipt, the XrEventDataBaseHeader pointer should be type-cast to a pointer of the appropriate event data based on the `type` parameter.

Valid Usage (Implicit)

- `type` **must** be one of the following XrStructureType values: `XR_TYPE_EVENT_DATA_DISPLAY_REFRESH_RATE_CHANGED_FB`, `XR_TYPE_EVENT_DATA_EVENTS_LOST`, `XR_TYPE_EVENT_DATA_INSTANCE_LOSS_PENDING`, `XR_TYPE_EVENT_DATA_INTERACTION_PROFILE_CHANGED`, `XR_TYPE_EVENT_DATA_MAIN_SESSION_VISIBILITY_CHANGED_EXTX`, `XR_TYPE_EVENT_DATA_PERF_SETTINGS_EXT`, `XR_TYPE_EVENT_DATA_REFERENCE_SPACE_CHANGE_PENDING`, `XR_TYPE_EVENT_DATA_SESSION_STATE_CHANGED`, `XR_TYPE_EVENT_DATA_VISIBILITY_MASK_CHANGED_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

The XrEventDataBuffer is a structure passed to xrPollEvent large enough to contain any returned event data element. The maximum size is specified by XR_MAX_EVENT_DATA_SIZE.

It is sufficient to clear the `type` and `next` parameters of an XrEventDataBuffer when passing it as an input to xrPollEvent.

An XrEventDataBuffer may be type-cast to an XrEventDataBaseHeader pointer or a pointer to any other appropriate event data based on the `type` parameter.

```
typedef struct XrEventDataBuffer {
    XrStructureType    type;
    const void*        next;
    uint8_t            varying[4000];
} XrEventDataBuffer;
```

Parameter Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `varying` is a fixed sized output buffer big enough to hold returned data elements for all specified event data types.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_EVENT_DATA_BUFFER`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

XR_MAX_EVENT_DATA_SIZE is the maximum size of an XrEventDataBuffer.

```
#define XR_MAX_EVENT_DATA_SIZE sizeof(XrEventDataBuffer)
```

XrEventDataEventsLost

The XrEventDataEventsLost structure is defined as:

```
typedef struct XrEventDataEventsLost {
    XrStructureType    type;
    const void*        next;
    uint32_t           lostEventCount;
} XrEventDataEventsLost;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `lostEventCount` is the number of events which have overflowed since the last call to xrPollEvent.

Receiving the XrEventDataEventsLost event structure indicates that the event queue overflowed and some events were removed at the position within the queue at which this event was found.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_EVENT_DATA_EVENTS_LOST`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

Other event structures are defined in later chapters in the context where their definition is most relevant.

2.21. System resource lifetime

The creator of an underlying system resource is responsible for ensuring the resource's lifetime matches the lifetime of the associated OpenXR handle.

Resources passed as inputs from the application to the runtime when creating an OpenXR handle **should** not be freed while that handle is valid. A runtime **must** not free resources passed as inputs or decrease their reference counts (if applicable) from the initial value. For example, the graphics device handle (or pointer) passed in to xrCreateSession in `XrGraphicsBinding*` structure **should** be kept alive when the corresponding XrSession handle is valid, and **should** be freed by the application after the XrSession handle is destroyed.

Resources created by the runtime should not be freed by the application, and the application **should** maintain the same reference count (if applicable) at the destruction of the OpenXR handle as it had at its creation. For example, the `ID3D*Texture2D` objects in the `XrSwapchainImageD3D*` are created by the runtime and associated with the lifetime of the XrSwapchain handle. The application **should** not keep additional reference counts on any `ID3D*Texture2D` objects past the lifetime of the XrSwapchain handle, or make extra reference count decrease after destroying the XrSwapchain handle.

Chapter 3. API Initialization

Before using an OpenXR runtime, an application **must** initialize it by creating a XrInstance object. The following functions are useful for gathering information about the API layers and extensions installed on the system and creating the instance.

Instance Creation Functions

- xrEnumerateApiLayerProperties
- xrEnumerateInstanceExtensionProperties
- xrCreateInstance

xrEnumerateApiLayerProperties and xrEnumerateInstanceExtensionProperties **can** be called before calling xrCreateInstance.

3.1. Exported Functions

A dynamically linked library (.dll or .so) that implements the API loader **must** export all core OpenXR API functions. However, the application **can** gain access to extension functions by obtaining pointers to these functions through the use of xrGetInstanceProcAddr.

3.2. Function Pointers

Function pointers for all OpenXR functions **can** be obtained with the function xrGetInstanceProcAddr.

```
XrResult xrGetInstanceProcAddr(
    XrInstance                                  instance,
    const char*                                 name,
    PFN_xrVoidFunction*                         function);
```

Parameter Descriptions

- `instance` is the instance that the function pointer will be compatible with, or `NULL` for functions not dependent on any instance.
- `name` is the name of the function to obtain.
- `function` is the address of the function pointer to get.

xrGetInstanceProcAddr itself is obtained in a platform- and loader- specific manner. Typically, the loader library will export this function as a function symbol, so applications **can** link against the loader library, or load it dynamically and look up the symbol using platform-specific APIs. Loaders **must** export function symbols for all core OpenXR functions. Because of this, applications that use only the core OpenXR functions have no need to use xrGetInstanceProcAddr.

Because an application **can** call xrGetInstanceProcAddr before creating an instance, xrGetInstanceProcAddr returns a valid function pointer when the `instance` parameter is XR_NULL_HANDLE and the `name` parameter is one of the following strings:

No Instance Required

- xrEnumerateInstanceExtensionProperties
- xrEnumerateApiLayerProperties
- xrCreateInstance

xrGetInstanceProcAddr **must** return `XR_ERROR_HANDLE_INVALID` if `name` is not one of the above strings and `instance` is XR_NULL_HANDLE. xrGetInstanceProcAddr **may** return `XR_ERROR_HANDLE_INVALID` if `name` is not one of the above strings and `instance` is invalid but not XR_NULL_HANDLE.

xrGetInstanceProcAddr **must** return `XR_ERROR_FUNCTION_UNSUPPORTED` if `instance` is a valid instance and the string specified in `name` is not the name of an OpenXR core or enabled extension function.

If `name` is the name of an extension function, then the result returned by xrGetInstanceProcAddr will depend upon how the `instance` was created. If `instance` was created with the related extension's name appearing in the XrInstanceCreateInfo::`enabledExtensionNames` array, then xrGetInstanceProcAddr returns a valid function pointer. If the related extension's name did not appear in the XrInstanceCreateInfo::`enabledExtensionNames` array during the creation of `instance`, then xrGetInstanceProcAddr returns `XR_ERROR_FUNCTION_UNSUPPORTED`. Because of this, function pointers returned by xrGetInstanceProcAddr using one XrInstance may not be valid when used with objects related to a different XrInstance.

The returned function pointer is of type PFN_xrVoidFunction, and must be cast to the type of the function being queried.

The table below defines the various use cases for xrGetInstanceProcAddr and return value ("fp" is "function pointer") for each case.

Table 3. xrGetInstanceProcAddr behavior

`instance` **parameter**	`name` **parameter**	**return value**
*	`NULL`	undefined
invalid instance	*	undefined

instance parameter	name parameter	return value
NULL	xrEnumerateInstanceExtensionProperties	fp
NULL	xrEnumerateApiLayerProperties	fp
NULL	xrCreateInstance	fp
NULL	* (any name not covered above)	NULL
instance	core OpenXR function	fp[1]
instance	enabled extension function for instance	fp[1]
instance	* (any name not covered above)	NULL

1

The returned function pointer **must** only be called with a handle (the first parameter) that is instance or a child of instance.

Valid Usage (Implicit)

- If instance is not XR_NULL_HANDLE, instance **must** be a valid XrInstance handle
- name **must** be a null-terminated UTF-8 string
- function **must** be a pointer to a PFN_xrVoidFunction value

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE

```
typedef void (XRAPI_PTR *PFN_xrVoidFunction)(void);
```

Parameter Descriptions

- no parameters.

PFN_xrVoidFunction is a generic function pointer type returned by queries, specifically those to xrGetInstanceProcAddr.

Chapter 4. Instance

```
XR_DEFINE_HANDLE(XrInstance)
```

An OpenXR instance is an object that allows an OpenXR application to communicate with an OpenXR runtime. The application accomplishes this communication by calling xrCreateInstance and receiving a handle to the resulting XrInstance object.

The XrInstance object stores and tracks OpenXR-related application state, without storing any such state in the application's global address space. This allows the application to create multiple instances as well as safely encapsulate the application's OpenXR state since this object is opaque to the application. OpenXR runtimes **may** limit the number of simultaneous XrInstance objects that may be created and used, but they **must** support the creation and usage of at least one XrInstance object per process.

Physically, this state **may** be stored in any of the OpenXR loader, OpenXR API layers or the OpenXR runtime components. The exact storage and distribution of this saved state is implementation-dependent, except where indicated by this specification.

The tracking of OpenXR state in the instance allows the streamlining of the API, where the intended instance is inferred from the highest ascendant of an OpenXR function's target object. For example, in:

```
myResult = xrEndFrame(mySession, &myEndFrameDescription);
```

the XrSession object was created from an XrInstance object. The OpenXR loader typically keeps track of the XrInstance that is the parent of the XrSession object in this example and directs the function to the runtime associated with that instance. This tracking of OpenXR objects eliminates the need to specify an XrInstance in every OpenXR function.

4.1. API Layers and Extensions

Additional functionality **may** be provided by API layers or extensions. An API layer **must** not add or modify the definition of OpenXR functions, while an extension **may** do so.

The set of API layers to enable is specified when creating an instance, and those API layers are able to intercept any functions dispatched to that instance or any of its child objects.

Example API layers **may** include (but are not limited to):

- an API layer to dump out OpenXR API calls
- an API layer to perform OpenXR validation

To determine what set of API layers are available, OpenXR provides the xrEnumerateApiLayerProperties function:

```
XrResult xrEnumerateApiLayerProperties(
    uint32_t                                    propertyCapacityInput,
    uint32_t*                                   propertyCountOutput,
    XrApiLayerProperties*                       properties);
```

Parameter Descriptions

- `propertyCapacityInput` is the capacity of the properties array, or 0 to indicate a request to retrieve the required capacity.
- `propertyCountOutput` is a pointer to the count of properties written, or a pointer to the required capacity in the case that propertyCapacityInput is 0.
- `properties` is a pointer to an array of XrApiLayerProperties structures, but **can** be `NULL` if propertyCapacityInput is 0.
- See the Buffer Size Parameters section for a detailed description of retrieving the required `properties` size.

The list of available layers may change at any time due to actions outside of the OpenXR runtime, so two calls to xrEnumerateApiLayerProperties with the same parameters **may** return different results, or retrieve different `propertyCountOutput` values or `properties` contents.

Once an instance has been created, the layers enabled for that instance will continue to be enabled and valid for the lifetime of that instance, even if some of them become unavailable for future instances.

Valid Usage (Implicit)

- `propertyCountOutput` **must** be a pointer to a `uint32_t` value
- If `propertyCapacityInput` is not `0`, `properties` **must** be a pointer to an array of `propertyCapacityInput` XrApiLayerProperties structures

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_SIZE_INSUFFICIENT

The XrApiLayerProperties structure is defined as:

```
typedef struct XrApiLayerProperties {
    XrStructureType    type;
    void*              next;
    char               layerName[XR_MAX_API_LAYER_NAME_SIZE];
    XrVersion          specVersion;
    uint32_t           layerVersion;
    char               description[XR_MAX_API_LAYER_DESCRIPTION_SIZE];
} XrApiLayerProperties;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- layerName is a string specifying the name of the API layer. Use this name in the XrInstanceCreateInfo::enabledApiLayerNames array to enable this API layer for an instance.
- specVersion is the API version the API layer was written to, encoded as described in the API Version Numbers and Semantics section.
- layerVersion is the version of this API layer. It is an integer, increasing with backward compatible changes.
- description is a string providing additional details that **can** be used by the application to identify the API layer.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_API_LAYER_PROPERTIES`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

To enable a layer, the name of the layer **should** be added to the `enabledApiLayerNames` member of XrInstanceCreateInfo when creating an XrInstance.

Loader implementations **may** provide mechanisms outside this API for enabling specific API layers. API layers enabled through such a mechanism are implicitly enabled, while API layers enabled by including the API layer name in XrInstanceCreateInfo::`enabledApiLayerNames` are explicitly enabled. Except where otherwise specified, implicitly enabled and explicitly enabled API layers differ only in the way they are enabled. Explicitly enabling an API layer that is implicitly enabled has no additional effect.

Instance extensions are able to affect the operation of the instance and any of its child objects. As stated earlier, extensions can expand the OpenXR API and provide new functions or augment behavior.

Examples of extensions **may** be (but are not limited to):

Extension Examples

- an extension to include OpenXR functions to work with a new graphics API
- an extension to expose debug information via a callback

The application can determine the available instance extensions by calling xrEnumerateInstanceExtensionProperties:

```
XrResult xrEnumerateInstanceExtensionProperties(
    const char*                                 layerName,
    uint32_t                                    propertyCapacityInput,
    uint32_t*                                   propertyCountOutput,
    XrExtensionProperties*                      properties);
```

Parameter Descriptions

- `layerName` is either `NULL` or a pointer to a string naming the API layer to retrieve extensions from, as returned by xrEnumerateApiLayerProperties.
- `propertyCapacityInput` is the capacity of the properties array, or `0` to indicate a request to retrieve the required capacity.
- `propertyCountOutput` is a pointer to the count of properties written, or a pointer to the required capacity in the case that `propertyCapacityInput` is `0`.
- `properties` is a pointer to an array of XrExtensionProperties structures, but **can** be `NULL` if `propertyCapacityInput` is `0`.
- See the Buffer Size Parameters section for a detailed description of retrieving the required `properties` size.

If `properties` is `NULL`, then the number of extensions properties available is returned in `propertyCountOutput`. Otherwise, `propertyCountInput` must point to a variable set by the user to the number of elements in the `properties` array. If `propertyCountInput` is less than the number of extension properties available, the contents of `properties` will be undefined. If `propertyCountInput` is smaller than the number of extensions available, the runtime **must** return the failure code `XR_ERROR_SIZE_INSUFFICIENT` and the contents of `properties` are undefined.

Because the list of available layers may change externally between calls to xrEnumerateInstanceExtensionProperties, two calls **may** retrieve different results if a `layerName` is available in one call but not in another. The extensions supported by a layer may also change between two calls, e.g. if the layer implementation is replaced by a different version between those calls.

Valid Usage (Implicit)

- If `layerName` is not `NULL`, `layerName` **must** be a null-terminated UTF-8 string
- `propertyCountOutput` **must** be a pointer to a `uint32_t` value
- If `propertyCapacityInput` is not `0`, `properties` **must** be a pointer to an array of `propertyCapacityInput` XrExtensionProperties structures

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_API_LAYER_NOT_PRESENT
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_SIZE_INSUFFICIENT
- XR_ERROR_RUNTIME_UNAVAILABLE

The XrExtensionProperties structure is defined as:

```
typedef struct XrExtensionProperties {
    XrStructureType    type;
    void*              next;
    char               extensionName[XR_MAX_EXTENSION_NAME_SIZE];
    uint32_t           extensionVersion;
} XrExtensionProperties;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `extensionName` is a `NULL` terminated string specifying the name of the extension.
- `extensionVersion` is the version of this extension. It is an integer, incremented with backward compatible changes.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_EXTENSION_PROPERTIES`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

4.2. Instance Lifecycle

The xrCreateInstance function is defined as:

```
XrResult xrCreateInstance(
    const XrInstanceCreateInfo*                 createInfo,
    XrInstance*                                 instance);
```

Parameter Descriptions

- `createInfo` points to an instance of XrInstanceCreateInfo controlling creation of the instance.
- `instance` points to an XrInstance handle in which the resulting instance is returned.

xrCreateInstance creates the XrInstance, then enables and initializes global API layers and extensions requested by the application. If an extension is provided by an API layer, both the API layer and extension **must** be specified at xrCreateInstance time. If a specified API layer cannot be found, no XrInstance will be created and the function will return `XR_ERROR_API_LAYER_NOT_PRESENT`. Likewise, if a specified extension cannot be found, the call **must** return `XR_ERROR_EXTENSION_NOT_PRESENT` and no XrInstance will be created. Additionally, some runtimes **may** limit the number of concurrent instances that may be in use. If the application attempts to create more instances than a runtime can simultaneously support, xrCreateInstance **may** return `XR_ERROR_LIMIT_REACHED`.

If the XrApplicationInfo::`applicationName` is the empty string the runtime **must** return `XR_ERROR_NAME_INVALID`.

If the XrInstanceCreateInfo structure contains a platform-specific extension for a platform other than the target platform, `XR_ERROR_INITIALIZATION_FAILED` **may** be returned. If a mandatory platform-specific extension is defined for the target platform but no matching extension struct is provided in XrInstanceCreateInfo the runtime **must** return `XR_ERROR_INITIALIZATION_FAILED`.

Valid Usage (Implicit)

- `createInfo` **must** be a pointer to a valid XrInstanceCreateInfo structure
- `instance` **must** be a pointer to an XrInstance handle

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_LIMIT_REACHED
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_INITIALIZATION_FAILED
- XR_ERROR_API_VERSION_UNSUPPORTED
- XR_ERROR_API_LAYER_NOT_PRESENT
- XR_ERROR_EXTENSION_NOT_PRESENT
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_NAME_INVALID
- XR_ERROR_RUNTIME_UNAVAILABLE

The XrInstanceCreateInfo structure is defined as:

```
typedef struct XrInstanceCreateInfo {
    XrStructureType          type;
    const void*              next;
    XrInstanceCreateFlags    createFlags;
    XrApplicationInfo        applicationInfo;
    uint32_t                 enabledApiLayerCount;
    const char* const*       enabledApiLayerNames;
    uint32_t                 enabledExtensionCount;
    const char* const*       enabledExtensionNames;
} XrInstanceCreateInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `createFlags` is a bitmask of XrInstanceCreateFlags that identifies options that apply to the creation.
- `applicationInfo` is an instance of XrApplicationInfo. This information helps runtimes recognize behavior inherent to classes of applications. XrApplicationInfo is defined in detail below.
- `enabledApiLayerCount` is the number of global API layers to enable.
- `enabledApiLayerNames` is a pointer to an array of `enabledApiLayerCount` strings containing the names of API layers to enable for the created instance. See the API Layers And Extensions section for further details.
- `enabledExtensionCount` is the number of global extensions to enable.
- `enabledExtensionNames` is a pointer to an array of `enabledExtensionCount` strings containing the names of extensions to enable.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_INSTANCE_CREATE_INFO`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrDebugUtilsMessengerCreateInfoEXT, XrInstanceCreateInfoAndroidKHR
- `createFlags` **must** be `0`
- `applicationInfo` **must** be a valid XrApplicationInfo structure
- If `enabledApiLayerCount` is not `0`, `enabledApiLayerNames` **must** be a pointer to an array of `enabledApiLayerCount` null-terminated UTF-8 strings
- If `enabledExtensionCount` is not `0`, `enabledExtensionNames` **must** be a pointer to an array of `enabledExtensionCount` null-terminated UTF-8 strings

The XrInstanceCreateFlags include:

```
// Flag bits for XrInstanceCreateFlags
```

There are currently no instance creation flags. This is reserved for future use.

The XrApplicationInfo structure is defined as:

```
typedef struct XrApplicationInfo {
    char          applicationName[XR_MAX_APPLICATION_NAME_SIZE];
    uint32_t      applicationVersion;
    char          engineName[XR_MAX_ENGINE_NAME_SIZE];
    uint32_t      engineVersion;
    XrVersion     apiVersion;
} XrApplicationInfo;
```

Member Descriptions

- `applicationName` is a non-empty string containing the name of the application.
- `applicationVersion` is an unsigned integer variable containing the developer-supplied version number of the application.
- `engineName` is a string containing the name of the engine (if any) used to create the application. It may be empty to indicate no specified engine.
- `engineVersion` is an unsigned integer variable containing the developer-supplied version number of the engine used to create the application. May be zero to indicate no specified engine.
- `apiVersion` is the version of this API against which the application will run, encoded as described in the API Version Numbers and Semantics section. If the runtime does not support the requested `apiVersion` it **must** return `XR_ERROR_API_VERSION_UNSUPPORTED`.

Valid Usage (Implicit)

- `applicationName` **must** be a null-terminated UTF-8 string whose length is less than or equal to `XR_MAX_APPLICATION_NAME_SIZE`
- `engineName` **must** be a null-terminated UTF-8 string whose length is less than or equal to `XR_MAX_ENGINE_NAME_SIZE`

> *Note*
>
> When using the OpenXR API to implement a reusable engine that will be used by many applications, `engineName` **should** be set to a unique string that identifies the engine, and `engineVersion` **should** encode a representation of the engine's version. This way, all applications that share this engine version will provide the same `engineName` and `engineVersion` to the runtime. The engine **should** then enable individual applications to choose their specific `applicationName` and `applicationVersion`, enabling one application to be distinguished from another application.
>
> When using the OpenXR API to implement an individual application without a shared engine, the input `engineName` **should** be left empty and `engineVersion` **should** be set to 0. The `applicationName` **should** then be filled in with a unique string that identifies the app and the `applicationVersion` **should** encode a representation of the application's version.

The xrDestroyInstance function is defined as:

```
XrResult xrDestroyInstance(
    XrInstance                                  instance);
```

The xrDestroyInstance function is used to destroy an XrInstance.

Parameter Descriptions

- `instance` is the handle to the instance to destroy.

XrInstance handles are destroyed using xrDestroyInstance. When an XrInstance is destroyed, all handles that are children of that XrInstance are also destroyed.

Valid Usage (Implicit)

- `instance` **must** be a valid XrInstance handle

Thread Safety

- Access to `instance`, and any child handles, **must** be externally synchronized

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID

4.3. Instance Information

The xrGetInstanceProperties function provides information about the instance and the associated runtime.

```
XrResult xrGetInstanceProperties(
    XrInstance                                  instance,
    XrInstanceProperties*                       instanceProperties);
```

Parameter Descriptions

- instance is a handle to an XrInstance previously created with xrCreateInstance.
- instanceProperties points to an XrInstanceProperties which describes the instance.

The instanceProperties parameter **must** be filled out by the runtime in response to this call, with information as defined in XrInstanceProperties.

Valid Usage (Implicit)

- instance **must** be a valid XrInstance handle
- instanceProperties **must** be a pointer to an XrInstanceProperties structure

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_VALIDATION_FAILURE

The XrInstanceProperties structure is defined as:

```
typedef struct XrInstanceProperties {
    XrStructureType    type;
    void*              next;
    XrVersion          runtimeVersion;
    char               runtimeName[XR_MAX_RUNTIME_NAME_SIZE];
} XrInstanceProperties;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- runtimeVersion is the runtime's version (not necessarily related to an OpenXR API version), expressed in the format of XR_MAKE_VERSION.
- runtimeName is the name of the runtime.

Valid Usage (Implicit)

- type **must** be XR_TYPE_INSTANCE_PROPERTIES
- next **must** be NULL or a valid pointer to the next structure in a structure chain

4.4. Platform-Specific Instance Creation

Some amount of data required for instance creation is exposed through chained structures defined in extensions. These structures may be **optional** or even **required** for instance creation on specific platforms, but not on other platforms. Separating off platform-specific functionality into extension structures prevents the primary XrInstanceCreateInfo structure from becoming too bloated with unnecessary information.

See the List of Extensions appendix for the list of available extensions and their related structures. These structures expand the XrInstanceCreateInfo parent struct using the XrInstanceCreateInfo::`next` member. The specific list of structures that may be used for extending XrInstanceCreateInfo::`next` can be found in the "Valid Usage (Implicit)" block immediately following the definition of the structure.

4.4.1. The Instance Lost Error

The `XR_ERROR_INSTANCE_LOST` error indicates that the XrInstance has become unusable. This **can** happen if a critical runtime process aborts, if the connection to the runtime is otherwise no longer available, or if the runtime encounters an error during any function execution which prevents it from being able to support further function execution. Once `XR_ERROR_INSTANCE_LOST` is first returned, it **must** henceforth be returned by all non-destroy functions that involve an XrInstance or child handle type until the instance is destroyed. Applications **must** destroy the XrInstance. Applications **may** then attempt to continue by recreating all relevant OpenXR objects, starting with a new XrInstance. A runtime **may** generate an XrEventDataInstanceLossPending event when instance loss is detected.

4.4.2. XrEventDataInstanceLossPending

```
typedef struct XrEventDataInstanceLossPending {
    XrStructureType    type;
    const void*        next;
    XrTime             lossTime;
} XrEventDataInstanceLossPending;
```

Receiving the XrEventDataInstanceLossPending event structure indicates that the application is about to lose the indicated XrInstance at the indicated `lossTime` in the future. The application should call xrDestroyInstance and relinquish any instance-specific resources. This typically occurs to make way for a replacement of the underlying runtime, such as via a software update.

After the application has destroyed all of its instances and their children and waited past the specified time, it may then re-try xrCreateInstance in a loop waiting for whatever maintenance the runtime is performing to complete. The runtime will return `XR_ERROR_INSTANCE_LOST` from xrCreateInstance as long as it is unable to create the instance. Once the runtime has returned and is able to continue, it **must** resume returning `XR_SUCCESS` from xrCreateInstance if valid data is passed in.

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `lossTime` is the absolute time at which the indicated instance will be considered lost and become unusable.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_EVENT_DATA_INSTANCE_LOSS_PENDING`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

4.5. Instance Enumerated Type String Functions

Applications often want to turn certain enum values from the runtime into strings for use in log messages, to be localized in UI, or for various other reasons. OpenXR provides functions that turn common enum types into UTF-8 strings for use in applications.

```
XrResult xrResultToString(
    XrInstance                                  instance,
    XrResult                                    value,
    char                                        buffer[XR_MAX_RESULT_STRING_SIZE]);
```

Parameter Descriptions

- `instance` is the handle of the instance to ask for the string.
- `value` is the XrResult value to turn into a string.
- `buffer` is the buffer that will be used to return the string in.

Returns the text version of the provided XrResult value as a UTF-8 string.

In all cases the returned string **must** be one of:

Result String Return Values

- The literal string defined for the provide numeric value in the core spec or extension. (e.g. the value 0 results in the string XR_SUCCESS)
- XR_UNKNOWN_SUCCESS_ concatenated with the positive result number expressed as a decimal number.
- XR_UNKNOWN_FAILURE_ concatenated with the negative result number expressed as a decimal number.

Valid Usage (Implicit)

- instance **must** be a valid XrInstance handle
- value **must** be a valid XrResult value
- buffer **must** be a character array of length XR_MAX_RESULT_STRING_SIZE

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_VALIDATION_FAILURE

The xrStructureTypeToString function is defined as:

```
XrResult xrStructureTypeToString(
    XrInstance                                  instance,
    XrStructureType                             value,
    char                                        buffer[XR_MAX_STRUCTURE_NAME_SIZE]);
```

Parameter Descriptions

- `instance` is the handle of the instance to ask for the string.
- `value` is the XrStructureType value to turn into a string.
- `buffer` is the buffer that will be used to return the string in.

Returns the text version of the provided XrStructureType value as a UTF-8 string.

In all cases the returned string **must** be one of:

Structure Type String Return Values

- The literal string defined for the provide numeric value in the core spec or extension. (e.g. the value of `XR_TYPE_INSTANCE_CREATE_INFO` results in the string `XR_TYPE_INSTANCE_CREATE_INFO`)
- `XR_UNKNOWN_STRUCTURE_TYPE_` concatenated with the structure type number expressed as a decimal number.

Valid Usage (Implicit)

- `instance` **must** be a valid XrInstance handle
- `value` **must** be a valid XrStructureType value
- `buffer` **must** be a character array of length `XR_MAX_STRUCTURE_NAME_SIZE`

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_VALIDATION_FAILURE`

Chapter 5. System

This API separates the concept of physical systems of XR devices from the logical objects that applications interact with directly. A system represents a collection of related devices in the runtime, often made up of several individual hardware components working together to enable XR experiences. An `XrSystemId` is returned by xrGetSystem representing the system of devices the runtime will use to support a given form factor. Each system may include: a VR/AR display, various forms of input (gamepad, touchpad, motion controller), and other trackable objects.

The application uses the system to create a session, which can then be used to accept input from the user and output rendered frames. The application also provides a default set of bindings from its actions to any number of input sources. The runtime **may** use this action information to activate only a subset of devices and avoid wasting resources on devices that are not in use. Exactly which devices are active once an XR system is selected will depend on the features provided by the runtime, and **may** vary from runtime to runtime. For example, a runtime that is capable of mapping from one tracking system's space to another's **may** support devices from multiple tracking systems simultaneously.

5.1. Form Factors

The first step in selecting a system is for the application to request its desired **form factor**. The form factor defines how the display(s) moves in the environment relative to the user's head and how the user will interact with the XR experience. A runtime **may** support multiple form factors, such as on a mobile phone that supports both slide-in VR headset experiences and handheld AR experiences.

While an application's core XR rendering may span across form factors, its user interface will often be written to target a particular form factor, requiring explicit tailoring to function well on other form factors. For example, screen-space UI designed for a handheld phone will produce an uncomfortable experience for users if presented in screen-space on an AR headset.

```
typedef enum XrFormFactor {
    XR_FORM_FACTOR_HEAD_MOUNTED_DISPLAY = 1,
    XR_FORM_FACTOR_HANDHELD_DISPLAY = 2,
    XR_FORM_FACTOR_MAX_ENUM = 0x7FFFFFFF
} XrFormFactor;
```

The predefined form factors which **may** be supported by OpenXR runtimes are:

Enumerant Descriptions

- XR_FORM_FACTOR_HEAD_MOUNTED_DISPLAY. The tracked display is attached to the user's head. The user cannot touch the display itself. A VR headset would be an example of this form factor.
- XR_FORM_FACTOR_HANDHELD_DISPLAY. The tracked display is held in the user's hand, independent from the user's head. The user **may** be able to touch the display, allowing for screen-space UI. A mobile phone running an AR experience using pass-through video would be an example of this form factor.

5.2. Getting the XrSystemId

```
XR_DEFINE_ATOM(XrSystemId)
```

An XrSystemId is an opaque atom used by the runtime to identify a system. The value XR_NULL_SYSTEM_ID is considered an invalid system.

```
#define XR_NULL_SYSTEM_ID 0
```

The only XrSystemId value defined to be constant across all instances is the invalid system XR_NULL_SYSTEM_ID. No supported system is associated with XR_NULL_SYSTEM_ID. Unless explicitly permitted, it **should** not be passed to API calls or used as a structure attribute when a valid XrSystemId is required.

The xrGetSystem function is defined as:

```
XrResult xrGetSystem(
    XrInstance                                  instance,
    const XrSystemGetInfo*                      getInfo,
    XrSystemId*                                 systemId);
```

Parameter Descriptions

- `instance` is the handle of the instance from which to get the information.
- `getInfo` is a pointer to an XrSystemGetInfo structure containing the application's requests for a system.
- `systemId` is the returned `XrSystemId`.

To get an `XrSystemId`, an application specifies its desired form factor to xrGetSystem and gets the runtime's `XrSystemId` associated with that configuration.

If the form factor is supported but temporarily unavailable, xrGetSystem **must** return `XR_ERROR_FORM_FACTOR_UNAVAILABLE`. A runtime **may** return `XR_SUCCESS` on a subsequent call for a form factor it previously returned `XR_ERROR_FORM_FACTOR_UNAVAILABLE`. For example, connecting or warming up hardware might cause an unavailable form factor to become available.

Valid Usage (Implicit)

- `instance` **must** be a valid XrInstance handle
- `getInfo` **must** be a pointer to a valid XrSystemGetInfo structure
- `systemId` **must** be a pointer to an `XrSystemId` value

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_FORM_FACTOR_UNAVAILABLE`
- `XR_ERROR_FORM_FACTOR_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`

The XrSystemGetInfo structure is defined as:

```
typedef struct XrSystemGetInfo {
    XrStructureType    type;
    const void*        next;
    XrFormFactor       formFactor;
} XrSystemGetInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `formFactor` is the XrFormFactor requested by the application.

The XrSystemGetInfo structure specifies attributes about a system as desired by an application.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_SYSTEM_GET_INFO`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `formFactor` **must** be a valid XrFormFactor value

```
XrInstance instance; // previously initialized

XrSystemGetInfo system_get_info;
memset(&system_get_info, 0, sizeof(system_get_info));
system_get_info.type = XR_TYPE_SYSTEM_GET_INFO;
system_get_info.formFactor = XR_FORM_FACTOR_HEAD_MOUNTED_DISPLAY;

XrSystemId systemId;
CHK_XR(xrGetSystem(instance, &system_get_info, &systemId));

// create session
// create swapchains
// begin session

// main loop

// end session
// destroy session

// no access to hardware after this point
```

5.3. System Properties

The xrGetSystemProperties function is defined as:

```
XrResult xrGetSystemProperties(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    XrSystemProperties*                         properties);
```

Parameter Descriptions

- `instance` is the instance from which `systemId` was retrieved.
- `systemId` is the `XrSystemId` whose properties will be queried.
- `properties` points to an instance of the XrSystemProperties structure, that will be filled with returned information.

An application **can** call xrGetSystemProperties to retrieve information about the system such as vendor ID, system name, and graphics and tracking properties.

Valid Usage (Implicit)

- `instance` **must** be a valid XrInstance handle
- `properties` **must** be a pointer to an XrSystemProperties structure

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_OUT_OF_MEMORY`
- `XR_ERROR_SYSTEM_INVALID`
- `XR_ERROR_VALIDATION_FAILURE`

The XrSystemProperties structure is defined as:

```
typedef struct XrSystemProperties {
    XrStructureType               type;
    void*                         next;
    XrSystemId                    systemId;
    uint32_t                      vendorId;
    char                          systemName[XR_MAX_SYSTEM_NAME_SIZE];
    XrSystemGraphicsProperties    graphicsProperties;
    XrSystemTrackingProperties    trackingProperties;
} XrSystemProperties;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- vendorId is a unique identifier for the vendor of the system.
- systemId is the XrSystemId identifying the system.
- systemName is a string containing the name of the system.
- graphicsProperties is an XrSystemGraphicsProperties structure specifying the system graphics properties.
- trackingProperties is an XrSystemTrackingProperties structure specifying system tracking properties.

Valid Usage (Implicit)

- type **must** be XR_TYPE_SYSTEM_PROPERTIES
- next **must** be NULL or a valid pointer to the next structure in a structure chain. See also: XrSystemEyeGazeInteractionPropertiesEXT, XrSystemFoveatedRenderingPropertiesVARJO, XrSystemHandTrackingMeshPropertiesMSFT, XrSystemHandTrackingPropertiesEXT

The XrSystemGraphicsProperties structure is defined as:

```
typedef struct XrSystemGraphicsProperties {
    uint32_t    maxSwapchainImageHeight;
    uint32_t    maxSwapchainImageWidth;
    uint32_t    maxLayerCount;
} XrSystemGraphicsProperties;
```

Member Descriptions

- `maxSwapchainImageHeight` is the maximum swapchain image pixel height supported by this system.
- `maxSwapchainImageWidth` is the maximum swapchain image pixel width supported by this system.
- `maxLayerCount` is the maximum number of composition layers supported by this system. The runtime **must** support at least `XR_MIN_COMPOSITION_LAYERS_SUPPORTED` layers.

The XrSystemTrackingProperties structure is defined as:

```
typedef struct XrSystemTrackingProperties {
    XrBool32    orientationTracking;
    XrBool32    positionTracking;
} XrSystemTrackingProperties;
```

Member Descriptions

- `orientationTracking` is set to `XR_TRUE` to indicate the system supports orientational tracking of the view pose(s), `XR_FALSE` otherwise.
- `positionTracking` is set to `XR_TRUE` to indicate the system supports positional tracking of the view pose(s), `XR_FALSE` otherwise.

Chapter 6. Path Tree and Semantic Paths

OpenXR incorporates an internal *semantic path tree* model, also known as the *path tree*, with entities associated with nodes organized in a logical tree and referenced by path name strings structured like a filesystem path or URL. The path tree unifies a number of concepts used in this specification and a runtime **may** add additional nodes as implementation details. As a general design principle, the most application-facing paths **should** have semantic and hierarchical meaning in their name. Thus, these paths are often referred to as *semantic paths*. However, path names in the path tree model **may** not all have the same level or kind of semantic meaning.

In regular use in an application, path name strings are converted to instance-specific `XrPath` values which are used in place of path strings. The mapping between `XrPath` values and their corresponding path name strings **may** be considered to be tracked by the runtime in a one-to-one mapping in addition to the natural tree structure of the referenced entities. Runtimes **may** use any internal implementation that satisfies the requirements.

Formally, the runtime maintains an instance-specific bijective mapping between well-formed path name strings and valid `XrPath` (`uint64_t`) values. These `XrPath` values are only valid within a single XrInstance, and applications **must** not share these values between instances. Applications **must** instead use the string representation of a path in their code and configuration, and obtain the correct corresponding `XrPath` at runtime in each XrInstance. The term *path* or *semantic path* **may** refer interchangeably to either the path name string or its associated `XrPath` value within an instance when context makes it clear which type is being discussed.

Given that path trees are a unifying model in this specification, the entities referenced by paths **can** be of diverse types. For example, they **may** be used to represent physical device or sensor *components*, which **may** be of various *component types*. They **may** also be used to represent frames of reference that are understood by the application and the runtime, as defined by an XrSpace. Additionally, to permit runtime re-configuration and support hardware-independent development, any syntactically-valid path string **may** be used to retrieve a corresponding `XrPath` without error given sufficient resources, *even if* no logical or hardware entity currently corresponds to that path at the time of the call. Later retrieval of the associated path string of such an `XrPath` using xrPathToString **should** succeed if the other requirements of that call are met. However, using such an `XrPath` in a later call to any other API function **may** result in an error if no entity of the type required by the call is available at the path at that later time. A runtime **should** permit the entity referenced by a path to vary over time to naturally reflect varying system configuration and hardware availability.

6.1. Path Atom Type

```
XR_DEFINE_ATOM(XrPath)
```

The XrPath is an atom that connects an application with a single path, within the context of a single instance. There is a bijective mapping between well-formed path strings and atoms in use. This atom is used — in place of the path name string it corresponds to — to retrieve state and perform other operations.

As an XrPath is only shorthand for a well-formed path string, they have no explicit life cycle.

Lifetime is implicitly managed by the XrInstance. An XrPath **must** not be used unless it is received at execution time from the runtime in the context of a particular XrInstance. Therefore, with the exception of XR_NULL_PATH, XrPath values **must** not be specified as constant values in applications: the corresponding path string **should** be used instead. During the lifetime of a given XrInstance, the XrPath associated with that instance with any given well-formed path **must** not vary, and similarly the well-formed path string that corresponds to a given XrPath in that instance **must** not vary. An XrPath that is received from one XrInstance **may** not be used with another. Such an invalid use **may** be detected and result in an error being returned, or it **may** result in undefined behavior.

Well-written applications **should** typically use a small, bounded set of paths in practice. However, the runtime **should** support looking up the XrPath for a large number of path strings for maximum compatibility. Runtime implementers **should** keep in mind that applications supporting diverse systems **may** look up path strings in a quantity exceeding the number of non-empty entities predicted or provided by any one runtime's own path tree model, and this is not inherently an error. However, system resources are finite and thus runtimes **may** signal exhaustion of resources dedicated to these associations under certain conditions.

When discussing the behavior of runtimes at these limits, a *new* XrPath refers to an XrPath value that, as of some point in time, has neither been received by the application nor tracked internally by the runtime. In this case, since an application has not yet received the value of such an XrPath, the runtime has not yet made any assertions about its association with any path string. In this context, *new* only refers to the fact that the mapping has not necessarily been made constant for a given value/path string pair for the remaining life of the associated instance by being revealed to the application. It does not necessarily imply creation of the entity, if any, referred to by such a path. Similarly, it does not imply the absence of such an entity prior to that point. Entities in the path tree have varied lifetime that is independent from the duration of the mapping from path string to XrPath.

For flexibility, the runtime **may** internally track or otherwise make constant, in instance or larger scope, any mapping of a path string to an XrPath value even before an application would otherwise receive that value, thus making it no longer *new* by the above definition.

When the runtime's resources to track the path string-XrPath mapping are exhausted, and the application makes an API call that would have otherwise retrieved a *new* XrPath as defined above, the runtime **must** return XR_ERROR_PATH_COUNT_EXCEEDED. This includes both explicit calls to xrStringToPath as well as other calls that retrieve an XrPath in any other way.

The runtime **should** support creating as many paths as memory will allow and **must** return XR_ERROR_PATH_COUNT_EXCEEDED from relevant functions when no more can be created.

```
#define XR_NULL_PATH 0
```

The only XrPath value defined to be constant across all instances is the invalid path XR_NULL_PATH. No well-formed path string is associated with XR_NULL_PATH. Unless explicitly permitted, it **should** not be passed to API calls or used as a structure attribute when a valid XrPath is required.

6.2. Well-Formed Path Strings

Even though they look similar, semantic paths are not file paths. To avoid confusion with file path directory traversal conventions, many file path conventions are explicitly disallowed from well-formed path name strings.

A well-formed path name string **must** conform to the following rules:

- Path name strings **must** be constructed entirely from characters on the following list.
  - Lower case ASCII letters: a-z
  - Numeric digits: 0-9
  - Dash: -
  - Underscore: _
  - Period: .
  - Forward Slash: /
- Path name strings **must** start with a single forward slash character.
- Path name strings **must** not end with a forward slash character.
- Path name strings **must** not contain two or more adjacent forward slash characters.
- Path name strings **must** not contain two forward slash characters that are separated by only period characters.
- Path name strings **must** not contain only period characters following the final forward slash character in the string.
- The maximum string length for a path name string, including the terminating `\0` character, is defined by `XR_MAX_PATH_LENGTH`.

6.2.1. xrStringToPath

The xrStringToPath function is defined as:

```
XrResult xrStringToPath(
    XrInstance                                  instance,
    const char*                                 pathString,
    XrPath*                                     path);
```

Parameter Descriptions

- `instance` is an instance previously created.
- `pathString` is the path name string to retrieve the associated `XrPath` for.
- `path` is the output parameter, which **must** point to an `XrPath`. Given a well-formed path name string, this will be populated with an opaque value that is constant for that path string during the lifetime of that instance.

xrStringToPath retrieves the `XrPath` value for a well-formed path string. If such a value had not yet been assigned by the runtime to the provided path string in this XrInstance, one **must** be assigned at this point. All calls to this function with the same XrInstance and path string **must** retrieve the same `XrPath` value. Upon failure, xrStringToPath **must** return an appropriate XrResult, and **may** set the output parameter to XR_NULL_PATH. See Path Atom Type for the conditions under which an error **may** be returned when this function is given a valid XrInstance and a well-formed path string.

If the runtime's resources are exhausted and it cannot create the path, a return value of `XR_ERROR_PATH_COUNT_EXCEEDED` **must** be returned. If the application specifies a string that is not a well-formed path string, `XR_ERROR_PATH_FORMAT_INVALID` **must** be returned.

> A return value of `XR_SUCCESS` from xrStringToPath **may** not necessarily imply that the runtime has a component or other source of data that will be accessible through that semantic path. It only means that the path string supplied was well-formed and that the retrieved `XrPath` maps to the given path string within and during the lifetime of the XrInstance given.

Valid Usage (Implicit)

- `instance` **must** be a valid XrInstance handle
- `pathString` **must** be a null-terminated UTF-8 string
- `path` **must** be a pointer to an `XrPath` value

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_PATH_FORMAT_INVALID
- XR_ERROR_PATH_COUNT_EXCEEDED
- XR_ERROR_VALIDATION_FAILURE

6.2.2. xrPathToString

```
XrResult xrPathToString(
    XrInstance                                  instance,
    XrPath                                      path,
    uint32_t                                    bufferCapacityInput,
    uint32_t*                                   bufferCountOutput,
    char*                                       buffer);
```

Parameter Descriptions

- instance is an instance previously created.
- path is the valid XrPath value to retrieve the path string for.
- bufferCapacityInput is the capacity of the buffer, or 0 to indicate a request to retrieve the required capacity.
- bufferCountOutput is a pointer to the count of characters written (including the terminating '\0'), or a pointer to the required capacity in the case that bufferCapacityInput is 0.
- buffer is a pointer to an application-allocated buffer that will be filled with the semantic path string. It **can** be NULL if bufferCapacityInput is 0.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required buffer size.

xrPathToString retrieves the path name string associated with an XrPath, in the context of a given

XrInstance, in the form of a NULL terminated string placed into a *caller-allocated* buffer. Since the mapping between a well-formed path name string and an XrPath is bijective, there will always be exactly one string for each valid XrPath value. This can be useful if the calling application receives an XrPath value that they had not previously retrieved via xrStringToPath. During the lifetime of the given XrInstance, the path name string retrieved by this function for a given valid XrPath will not change. For invalid paths, including XR_NULL_PATH, XR_ERROR_PATH_INVALID **must** be returned.

Valid Usage (Implicit)

- instance **must** be a valid XrInstance handle
- bufferCountOutput **must** be a pointer to a uint32_t value
- If bufferCapacityInput is not 0, buffer **must** be a pointer to an array of bufferCapacityInput char values

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_PATH_INVALID
- XR_ERROR_SIZE_INSUFFICIENT
- XR_ERROR_VALIDATION_FAILURE

6.3. Reserved Paths

In order for some uses of semantic paths to work consistently across runtimes, it is necessary to standardize several paths and require each runtime to use the same paths or patterns of paths for certain classes of usage. Those paths are as follows.

6.3.1. /user paths

Some paths are used to refer to entities that are filling semantic roles in the system. These paths are all under the */user* subtree.

The reserved user paths are:

Reserved Semantic Paths

- */user/hand/left* represents the user's left hand. It might be tracked using a controller or other device in the user's left hand, or tracked without the user holding anything, e.g. using computer vision.
- */user/hand/right* represents the user's right hand in analog to the left hand.
- */user/head* represents inputs on the user's head, often from a device such as a head-mounted display. To reason about the user's head, see the `XR_REFERENCE_SPACE_TYPE_VIEW` reference space.
- */user/gamepad* is a two-handed gamepad device held by the user.
- */user/treadmill* is a treadmill or other locomotion-targeted input device.

Runtimes are not required to provide interaction at all of these paths. For instance, in a system with no hand tracking, only */user/head* would be active for interaction. In a system with only one controller, the runtime **may** provide access to that controller via either */user/hand/left* or */user/hand/right* as it deems appropriate.

The runtime **may** change the devices referred to by */user/hand/left* and */user/hand/right* at any time.

If more than two hand-held controllers or devices are active, the runtime **must** determine which two are accessible as */user/hand/left* and */user/hand/right*.

6.3.2. Input subpaths

Devices on the source side of the input system need to define paths for each component that can be bound to an action. This section describes the naming conventions for those input components. Runtimes **must** ignore input source paths that use identifiers and component names that do not appear in this specification or otherwise do not follow the pattern specified below.

Each input source path **must** match the following pattern:

- *…/input/*<identifier>[_<location>][/<component>]

Identifiers are often the label on the component or related to the type and location of the component.

When specifying a suggested binding there are several cases where the component part of the path can be determined automatically. See Suggested Bindings for more details.

See Interaction Profiles for examples of input subpaths.

Standard identifiers

- trackpad - A 2D input source that usually includes click and touch component.
- thumbstick - A small 2D joystick that is meant to be used with the user's thumb. These sometimes

include click and/or touch components.

- joystick - A 2D joystick that is meant to be used with the user's entire hand, such as a flight stick. These generally do not have click component, but might have touch components.
- trigger - A 1D analog input component that returns to a rest state when the user stops interacting with it. These sometime include touch and/or click components.
- throttle - A 1D analog input component that remains in position when the user stops interacting with it.
- trackball - A 2D relative input source. These sometimes include click components.
- pedal - A 1D analog input component that is similar to a trigger but meant to be operated by a foot
- system - A button with the specialised meaning that it enables the user to access system-level functions and UI. Input data from system buttons is generally used internally by runtimes and **may** not be available to applications.
- dpad_up, dpad_down, dpad_left, and dpad_right - A set of buttons arranged in a plus shape.
- diamond_up, diamond_down, diamond_left, and diamond_right - Gamepads often have a set of four buttons arranged in a diamond shape. The labels on those buttons vary from gamepad to gamepad, but their arrangement is consistent. These names are used for the A/B/X/Y buttons on a Xbox controller, and the square/cross/circle/triangle button on a PlayStation controller.
- a, b, x, y, start, home, end, select - Standalone buttons are named for their physical labels. These are the standard identifiers for such buttons. Extensions **may** add new identifiers as detailed in the next section. Groups of four buttons in a diamond shape **should** use the diamond-prefix names above instead of using the labels on the buttons themselves.
- volume_up, volume_down, mute_mic, play_pause, menu, view - Some other standard controls are often identified by icons. These are their standard names.
- thumbrest - Some controllers have a place for the user to rest their thumb.
- shoulder - A button that is usually pressed with the index finger and is often positioned above a trigger.
- squeeze - An input source that indicates that the user is squeezing their fist closed. This could be a simple button or act more like a trigger. Sources with this identifier **should** either follow button or trigger conventions for their components.
- wheel - A steering wheel.

Input sources whose orientation and/or position are tracked also expose pose identifiers.

Standard pose identifiers for tracked hands or motion controllers as represented by */user/hand/left* and */user/hand/right* are:

- grip - A pose that allows applications to reliably render a virtual object held in the user's hand, whether it is tracked directly or by a motion controller. The grip pose is defined as follows:
  - The grip position:

        - For tracked hands: The user’s palm centroid when closing the fist, at the surface of the palm.
        - For handheld motion controllers: A fixed position within the controller that generally lines up with the palm centroid when held by a hand in a neutral position. This position should be adjusted left or right to center the position within the controller’s grip.
    - The grip orientation’s +X axis: When you completely open your hand to form a flat 5-finger pose, the ray that is normal to the user’s palm (away from the palm in the left hand, into the palm in the right hand).
    - The grip orientation’s -Z axis: When you close your hand partially (as if holding the controller), the ray that goes through the center of the tube formed by your non-thumb fingers, in the direction of little finger to thumb.
    - The grip orientation’s +Y axis: orthogonal to +Z and +X using the right-hand rule.
- aim - A pose that allows applications to point in the world using the input source, according to the platform’s conventions for aiming with that kind of source. The aim pose is defined as follows:
    - For tracked hands: The ray that follows platform conventions for how the user aims at objects in the world with their entire hand, with +Y up, +X to the right, and -Z forward. The ray chosen will be runtime-dependent, for example, a ray emerging from the palm parallel to the forearm.
    - For handheld motion controllers: The ray that follows platform conventions for how the user targets objects in the world with the motion controller, with +Y up, +X to the right, and -Z forward. This is usually for applications that are rendering a model matching the physical controller, as an application rendering a virtual object in the user’s hand likely prefers to point based on the geometry of that virtual object. The ray chosen will be runtime-dependent, although this will often emerge from the frontmost tip of a motion controller.

Standard locations

When a single device contains multiple input sources that use the same identifier, a location suffix is added to create a unique identifier for that input source.

Standard locations are:

- left
- right
- left_upper
- left_lower
- right_upper
- right_lower
- upper
- lower

Standard components

Components are named for the specific boolean, scalar, or other value of the input source. Standard components are:

- click - A physical switch has been pressed by the user. This is valid for all buttons, and is common for trackpads, thumbsticks, triggers, and dpads. "click" components are always boolean.
- touch - The user has touched the input source. This is valid for all trackpads, and **may** be present for any other kind of input source if the device includes the necessary sensor. "touch" components are always boolean.
- force - A 1D scalar value that represents the user applying force to the input. It varies from 0 to 1, with 0 being the rest state. This is present for any input source with a force sensor.
- value - A 1D scalar value that varies from 0 to 1, with 0 being the rest state. This is present for triggers, throttles, and pedals. It **may** also be present for squeeze or other components.
- x, y - scalar components of 2D values. These vary in value from -1 to 1. These represent the 2D position of the input source with 0 being the rest state on each axis. -1 means all the way left for x axis or all the way down for y axis. +1 means all the way right for x axis or all the way up for y axis. x and y components are present for trackpads, thumbsticks, and joysticks.
- twist - Some sources, such as flight sticks, have a sensor that allows the user to twist the input left or right. For this component -1 means all the way left and 1 means all the way right.
- pose - The orientation and/or position of this input source. This component **may** exist for dedicated pose identifiers like grip and aim, or **may** be defined on other identifiers such as trackpad to let applications reason about the surface of that part.

Output paths

Many devices also have subpaths for output features such as haptics. The runtime **must** ignore output component paths that do not follow the pattern:

- *…/output/*<output_identifier>[_<location>]

Standard output identifiers are:

- haptic - A haptic element like an LRA (Linear Resonant Actuator) or vibration motor

Devices which contain multiple haptic elements with the same output identifier must use a location suffix as specified above.

6.3.3. Adding input sources via extensions

Extensions **may** enable input source path identifiers, output source path identifiers, and component names that are not included in the core specification, subject to the following conditions:

- EXT extensions **must** include the _ext suffix on any identifier or component name. E.g.

…/input/newidentifier_ext/newcomponent_ext

- Vendor extensions **must** include the vendor's tag as a suffix on any identifier or component name. E.g. *…/input/newidentifier_vendor/newcomponent_vendor* (where "vendor" is replaced with the vendor's actual extension tag.)
- Khronos (KHR) extensions **may** add undecorated identifier or component names.

These rules are in place to prevent extensions from adding first class undecorated names that become defacto standards. Runtimes **must** ignore input source paths that do not follow the restrictions above.

Extensions **may** also add new location suffixes, and **may** do so by adding a new identifier and location combination using the appropriate suffix. E.g. *…/input/newidentifier_newlocation_ext*

6.4. Interaction Profile Paths

An interaction profile path identifies a collection of buttons and other input sources in a physical arrangement to allow applications and runtimes to coordinate action bindings.

Interaction profile paths are of the form:

- */interaction_profiles/*<vendor_name>/<type_name>

6.4.1. Khronos Simple Controller Profile

Path: */interaction_profiles/khr/simple_controller*

Valid for user paths:

- */user/hand/left*
- */user/hand/right*

This interaction profile provides basic pose, button, and haptic support for applications with simple input needs. There is no hardware associated with the profile, and runtimes which support this profile **should** map the input paths provided to whatever the appropriate paths are on the actual hardware.

Supported component paths:

- *…/input/select/click*
- *…/input/menu/click*
- *…/input/grip/pose*
- *…/input/aim/pose*
- *…/output/haptic*

6.4.2. Google Daydream Controller Profile

Path: */interaction_profiles/google/daydream_controller*

Valid for user paths:

- */user/hand/left*
- */user/hand/right*

This interaction profile represents the input sources on the Google Daydream Controller.

Supported component paths:

- *…/input/select/click*
- *…/input/trackpad/x*
- *…/input/trackpad/y*
- *…/input/trackpad/click*
- *…/input/trackpad/touch*
- *…/input/grip/pose*
- *…/input/aim/pose*

6.4.3. HTC Vive Controller Profile

Path: */interaction_profiles/htc/vive_controller*

Valid for user paths:

- */user/hand/left*
- */user/hand/right*

This interaction profile represents the input sources and haptics on the Vive Controller.

Supported component paths:

- *…/input/system/click* (**may** not be available for application use)
- *…/input/squeeze/click*
- *…/input/menu/click*
- *…/input/trigger/click*
- *…/input/trigger/value*
- *…/input/trackpad/x*
- *…/input/trackpad/y*

- *…/input/trackpad/click*
- *…/input/trackpad/touch*
- *…/input/grip/pose*
- *…/input/aim/pose*
- *…/output/haptic*

6.4.4. HTC Vive Pro Profile

Path: */interaction_profiles/htc/vive_pro*

Valid for user paths:

- */user/head*

This interaction profile represents the input sources on the Vive Pro headset.

Supported component paths:

- *…/input/system/click* (**may** not be available for application use)
- *…/input/volume_up/click*
- *…/input/volume_down/click*
- *…/input/mute_mic/click*

6.4.5. Microsoft Mixed Reality Motion Controller Profile

Path: */interaction_profiles/microsoft/motion_controller*

Valid for user paths:

- */user/hand/left*
- */user/hand/right*

This interaction profile represents the input sources and haptics on the Microsoft Mixed Reality Controller.

Supported component paths:

- *…/input/menu/click*
- *…/input/squeeze/click*
- *…/input/trigger/value*
- *…/input/thumbstick/x*
- *…/input/thumbstick/y*

- .../input/thumbstick/click
- .../input/trackpad/x
- .../input/trackpad/y
- .../input/trackpad/click
- .../input/trackpad/touch
- .../input/grip/pose
- .../input/aim/pose
- .../output/haptic

6.4.6. Microsoft Xbox Controller Profile

Path: */interaction_profiles/microsoft/xbox_controller*

Valid for user paths:

- */user/gamepad*

This interaction profile represents the input sources and haptics on the Microsoft Xbox Controller.

Supported component paths:

- .../input/menu/click
- .../input/view/click
- .../input/a/click
- .../input/b/click
- .../input/x/click
- .../input/y/click
- .../input/dpad_down/click
- .../input/dpad_right/click
- .../input/dpad_up/click
- .../input/dpad_left/click
- .../input/shoulder_left/click
- .../input/shoulder_right/click
- .../input/thumbstick_left/click
- .../input/thumbstick_right/click
- .../input/trigger_left/value
- .../input/trigger_right/value

* *…/input/thumbstick_left/x*
* *…/input/thumbstick_left/y*
* *…/input/thumbstick_right/x*
* *…/input/thumbstick_right/y*
* *…/output/haptic_left*
* *…/output/haptic_right*
* *…/output/haptic_left_trigger*
* *…/output/haptic_right_trigger*

6.4.7. Oculus Go Controller Profile

Path: */interaction_profiles/oculus/go_controller*

Valid for user paths:

* */user/hand/left*
* */user/hand/right*

This interaction profile represents the input sources on the Oculus Go controller.

Supported component paths:

* *…/input/system/click* (**may** not be available for application use)
* *…/input/trigger/click*
* *…/input/back/click*
* *…/input/trackpad/x*
* *…/input/trackpad/y*
* *…/input/trackpad/click*
* *…/input/trackpad/touch*
* *…/input/grip/pose*
* *…/input/aim/pose*

6.4.8. Oculus Touch Controller Profile

Path: */interaction_profiles/oculus/touch_controller*

Valid for user paths:

* */user/hand/left*
* */user/hand/right*

This interaction profile represents the input sources and haptics on the Oculus Touch controller.

Supported component paths:

- On */user/hand/left* only:
  - *…/input/x/click*
  - *…/input/x/touch*
  - *…/input/y/click*
  - *…/input/y/touch*
  - *…/input/menu/click*
- On */user/hand/right* only:
  - *…/input/a/click*
  - *…/input/a/touch*
  - *…/input/b/click*
  - *…/input/b/touch*
  - *…/input/system/click* (**may** not be available for application use)
- *…/input/squeeze/value*
- *…/input/trigger/value*
- *…/input/trigger/touch*
- *…/input/thumbstick/x*
- *…/input/thumbstick/y*
- *…/input/thumbstick/click*
- *…/input/thumbstick/touch*
- *…/input/thumbrest/touch*
- *…/input/grip/pose*
- *…/input/aim/pose*
- *…/output/haptic*

6.4.9. Valve Index Controller Profile

Path: */interaction_profiles/valve/index_controller*

Valid for user paths:

- */user/hand/left*
- */user/hand/right*

This interaction profile represents the input sources and haptics on the Valve Index controller.

Supported component paths:

- *…/input/system/click* (**may** not be available for application use)
- *…/input/system/touch* (**may** not be available for application use)
- *…/input/a/click*
- *…/input/a/touch*
- *…/input/b/click*
- *…/input/b/touch*
- *…/input/squeeze/value*
- *…/input/squeeze/force*
- *…/input/trigger/click*
- *…/input/trigger/value*
- *…/input/trigger/touch*
- *…/input/thumbstick/x*
- *…/input/thumbstick/y*
- *…/input/thumbstick/click*
- *…/input/thumbstick/touch*
- *…/input/trackpad/x*
- *…/input/trackpad/y*
- *…/input/trackpad/force*
- *…/input/trackpad/touch*
- *…/input/grip/pose*
- *…/input/aim/pose*
- *…/output/haptic*

Chapter 7. Spaces

Across both virtual reality and augmented reality, XR applications have a core need to map the location of virtual objects to the corresponding real-world locations where they will be rendered. **Spaces** allow applications to explicitly create and specify the frames of reference in which they choose to track the real world, and then determine how those frames of reference move relative to one another over time.

```
XR_DEFINE_HANDLE(XrSpace)
```

Spaces are represented by XrSpace handles, which the application creates and then uses in API calls. Whenever an application calls a function that returns coordinates, it provides an XrSpace to specify the frame of reference in which those coordinates will be expressed. Similarly, when providing coordinates to a function, the application specifies which XrSpace the runtime should use to interpret those coordinates.

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

Runtimes whose tracking systems improve their understanding of the world over time **may** track spaces independently. For example, even though a `LOCAL` space and a `STAGE` space each map their origin to a static position in the world, a runtime with an inside-out tracking system **may** introduce slight adjustments to the origin of each space on a continuous basis to keep each origin in place.

Beyond well-known reference spaces, runtimes expose other independently-tracked spaces, such as a pose action space that tracks the pose of a motion controller over time.

When one or both spaces are tracking a dynamic object, passing in an updated time to xrLocateSpace each frame will result in an updated relative pose. For example, the location of the left hand's pose action space in the `STAGE` reference space will change each frame as the user's hand moves relative to the stage's predefined origin on the floor. In other XR APIs, it is common to report the "pose" of an object relative to some presumed underlying global space. This API is careful to not explicitly define such an underlying global space, because it does not apply to all systems. Some systems will support no `STAGE` space, while others may support a `STAGE` space that switches between various physical stages with dynamic availability. To satisfy this wide variability, "poses" are always described as the relationship between two spaces.

Some devices improve their understanding of the world as the device is used. The location returned by xrLocateSpace in later frames **may** change over time, even for spaces that track static objects, as either the target space or base space adjusts its origin.

Composition layers submitted by the application include an XrSpace for the runtime to use to position that layer over time. Composition layers whose XrSpace is relative to the `VIEW` reference space are implicitly "head-locked", even if they may not be "display-locked" for non-head-mounted form factors.

7.1. Reference Spaces

An XrSpace handle for a reference space is created using xrCreateReferenceSpace, by specifying the chosen reference space type and a pose within the natural reference frame defined for that reference space type.

Runtimes implement well-known reference spaces from XrReferenceSpaceType if they support tracking of that kind:

```
typedef enum XrReferenceSpaceType {
    XR_REFERENCE_SPACE_TYPE_VIEW = 1,
    XR_REFERENCE_SPACE_TYPE_LOCAL = 2,
    XR_REFERENCE_SPACE_TYPE_STAGE = 3,
    XR_REFERENCE_SPACE_TYPE_UNBOUNDED_MSFT = 1000038000,
    XR_REFERENCE_SPACE_TYPE_COMBINED_EYE_VARJO = 1000121000,
    XR_REFERENCE_SPACE_TYPE_MAX_ENUM = 0x7FFFFFFF
} XrReferenceSpaceType;
```

Available reference space types are indicated by xrEnumerateReferenceSpaces. Note that other spaces can be created as well, such as pose action spaces created by xrCreateActionSpace, which are not enumerated by that API.

Enumerant Descriptions

- XR_REFERENCE_SPACE_TYPE_VIEW. The VIEW space tracks the view origin used to generate view transforms for the primary viewer (or centroid of view origins if stereo), with +Y up, +X to the right, and -Z forward. This space points in the forward direction for the viewer without incorporating the user's eye orientation, and is not gravity-aligned.

  VIEW space is primarily useful when projecting from the user's perspective into another space to obtain a targeting ray, or when rendering small head-locked content such as a reticle. Content rendered in VIEW space will stay at a fixed point on head-mounted displays and may be uncomfortable to view if too large. To obtain the ideal view and projection transforms to use each frame for rendering world content, applications should call xrLocateViews instead of using this space.

  Runtimes **must** support this reference space.

- XR_REFERENCE_SPACE_TYPE_LOCAL. The LOCAL reference space establishes a world-locked origin, gravity-aligned to exclude pitch and roll, with +Y up, +X to the right, and -Z forward. This space locks in both its initial position and orientation, which the runtime **may** define to be either the initial position at application launch or some other calibrated zero position.

  LOCAL space is useful when an application needs to render **seated-scale** content that is not positioned relative to the physical floor.

  When a user needs to recenter LOCAL space, a runtime **may** offer some system-level recentering interaction that is transparent to the application, but which causes the current leveled head space to become the new LOCAL space. When such a recentering occurs, the runtime **must** queue the XrEventDataReferenceSpaceChangePending event, with the recentered LOCAL space origin only taking effect for xrLocateSpace or xrLocateViews calls whose XrTime parameter is greater than or equal to the changeTime provided in that event.

  When views, controllers or other spaces experience tracking loss relative to the LOCAL space, runtimes **should** continue to provide inferred or last-known position and orientation values. These inferred poses can, for example, be based on neck model updates, inertial dead reckoning, or a last-known position, so long as it is still reasonable for the application to use that pose. While a runtime is providing position data, it **must** continue to set XR_SPACE_LOCATION_POSITION_VALID_BIT and XR_VIEW_STATE_POSITION_VALID_BIT but it **can** clear XR_SPACE_LOCATION_POSITION_TRACKED_BIT and XR_VIEW_STATE_POSITION_TRACKED_BIT to indicate that the position is inferred or last-known in this way.

  When tracking is recovered, runtimes **should** snap the pose of other spaces back into position relative to the LOCAL space's original origin.

  Runtimes **must** support this reference space.

- XR_REFERENCE_SPACE_TYPE_STAGE. The STAGE reference space is a runtime-defined flat, rectangular space that is empty and can be walked around on. The origin is on the floor at the center of the rectangle, with +Y up, and the X and Z axes aligned with the rectangle edges. The runtime **may** not be able to locate spaces relative to the STAGE reference space if the user has not yet defined one within the runtime-specific UI. Applications can use xrGetReferenceSpaceBoundsRect to determine the extents of the STAGE reference space's XZ bounds rectangle, if defined.

STAGE space is useful when an application needs to render **standing-scale** content (no bounds) or **room-scale** content (with bounds) that is relative to the physical floor.

When the user redefines the origin or bounds of the current STAGE space, or the runtime otherwise switches to a new STAGE definition, the runtime **must** queue the XrEventDataReferenceSpaceChangePending event, with the new STAGE space origin only taking effect for xrLocateSpace or xrLocateViews calls whose XrTime parameter is greater than or equal to the changeTime provided in that event.

When views, controllers or other spaces experience tracking loss relative to the STAGE space, runtimes **should** continue to provide inferred or last-known position and orientation values. These inferred poses can, for example, be based on neck model updates, inertial dead reckoning, or a last-known position, so long as it is still reasonable for the application to use that pose. While a runtime is providing position data, it **must** continue to set XR_SPACE_LOCATION_POSITION_VALID_BIT and XR_VIEW_STATE_POSITION_VALID_BIT but it **can** clear XR_SPACE_LOCATION_POSITION_TRACKED_BIT and XR_VIEW_STATE_POSITION_TRACKED_BIT to indicate that the position is inferred or last-known in this way.

When tracking is recovered, runtimes **should** snap the pose of other spaces back into position relative to the STAGE space's original origin.

XR systems may have limited real world spatial ranges in which users can freely move around while remaining tracked. Applications may wish to query these boundaries and alter application behavior or content placement to ensure the user can complete the experience while remaining within the boundary. Applications **can** query this information using xrGetReferenceSpaceBoundsRect.

When called, xrGetReferenceSpaceBoundsRect **should** return the extents of a rectangle that is clear of obstacles down to the floor, allowing where the user can freely move while remaining tracked, if available for that reference space. The returned extent represents the dimensions of an axis-aligned bounding box where the XrExtent2Df::width and XrExtent2Df::height fields correspond to the X and Z axes of the provided space, with the extents centered at the origin of the space. Not all systems or spaces may support boundaries. If a runtime is unable to provide bounds for a given space, XR_SPACE_BOUNDS_UNAVAILABLE will be returned and all fields of bounds will be set to 0.

The returned extents are expressed relative to the natural origin of the provided XrReferenceSpaceType and **must** not incorporate any origin offsets specified by the application during calls to xrCreateReferenceSpace.

The runtime **must** return XR_ERROR_REFERENCE_SPACE_UNSUPPORTED if the XrReferenceSpaceType passed in createInfo is not supported by this session.

When a runtime will begin operating with updated space bounds, the runtime **must** queue a corresponding XrEventDataReferenceSpaceChangePending event.

```
XrResult xrGetReferenceSpaceBoundsRect(
    XrSession                                   session,
    XrReferenceSpaceType                        referenceSpaceType,
    XrExtent2Df*                                bounds);
```

Parameter Descriptions

- type is the XrStructureType of this structure.
- session is a handle to an XrSession previously created with xrCreateSession.
- referenceSpaceType is the reference space type whose bounds should be retrieved.
- bounds is the returned space extents.

Valid Usage (Implicit)

- session **must** be a valid XrSession handle
- referenceSpaceType **must** be a valid XrReferenceSpaceType value
- bounds **must** be a pointer to an XrExtent2Df structure

Return Codes

Success

- XR_SUCCESS
- XR_SPACE_BOUNDS_UNAVAILABLE
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_REFERENCE_SPACE_UNSUPPORTED

The XrEventDataReferenceSpaceChangePending event is sent to the application to notify it that the origin (and perhaps the bounds) of a reference space is changing. This may occur due to the user recentering the space explicitly, or the runtime otherwise switching to a different space definition.

The reference space change **must** only take effect for xrLocateSpace or xrLocateViews calls whose `XrTime` parameter is greater than or equal to the `changeTime` provided in that event. Runtimes **should** provide a `changeTime` to applications that allows for a deep render pipeline to present frames that are already in flight using the previous definition of the space. Runtimes **should** choose a `changeTime` that is midway between the `displayTime` of future frames to avoid threshold issues with applications that calculate future frame times using `displayPeriod`.

The `pose` provided here **must** only describe the change in the natural origin of the reference space and **must** not incorporate any origin offsets specified by the application during calls to xrCreateReferenceSpace. If the runtime does not know the location of the space's new origin relative to its previous origin, `poseValid` **must** be false, and the position and orientation of `poseInPreviousSpace` are undefined.

```
typedef struct XrEventDataReferenceSpaceChangePending {
    XrStructureType         type;
    const void*             next;
    XrSession               session;
    XrReferenceSpaceType    referenceSpaceType;
    XrTime                  changeTime;
    XrBool32                poseValid;
    XrPosef                 poseInPreviousSpace;
} XrEventDataReferenceSpaceChangePending;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `session` is the XrSession for which the reference space is changing.
- `referenceSpaceType` is the XrReferenceSpaceType that is changing.
- `changeTime` is the target `XrTime` after which xrLocateSpace or xrLocateViews will return values that respect this change.
- `poseValid` is true if the runtime can determine the `pose` of the new space in the previous space before the change.
- `poseInPreviousSpace` is an XrPosef defining the position and orientation of the new reference space's natural origin within the natural reference frame of its previous space.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_EVENT_DATA_REFERENCE_SPACE_CHANGE_PENDING`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `session` **must** be a valid XrSession handle
- `referenceSpaceType` **must** be a valid XrReferenceSpaceType value

7.2. Action Spaces

An XrSpace handle for a pose action is created using xrCreateActionSpace, by specifying the chosen pose action and a pose within the action's natural reference frame.

Runtimes support suggested pose action bindings to well-known user paths with *…/pose* subpaths if they support tracking for that particular identifier.

Some example well-known pose action paths:

- */user/hand/left/input/grip*
- */user/hand/left/input/aim*
- */user/hand/right/input/grip*
- */user/hand/right/input/aim*

For definitions of these well-known pose device paths, see the discussion of device input subpaths in the Semantic Paths chapter.

7.2.1. Action Spaces Lifetime

XrSpace handles created for a pose action **must** be unlocatable unless the action set that contains the corresponding pose action was set as active via the most recent xrSyncActions call. If the underlying device that is active for the action changes, the device this space is tracking **must** only change to track the new device when xrSyncActions is called.

If xrLocateSpace is called with an unlocatable action space, the implementation **must** return no position or orientation and both `XR_SPACE_LOCATION_POSITION_VALID_BIT` and `XR_SPACE_LOCATION_ORIENTATION_VALID_BIT` **must** be unset. If xrLocateViews is called with an unlocatable action space, the implementation **must** return no position or orientation and both `XR_VIEW_STATE_POSITION_VALID_BIT` and `XR_VIEW_STATE_ORIENTATION_VALID_BIT` **must** be unset.

7.3. Space Lifecycle

There are a small set of core APIs that allow applications to reason about reference spaces, action spaces, and their relative locations.

7.3.1. xrEnumerateReferenceSpaces

The xrEnumerateReferenceSpaces function is defined as:

```
XrResult xrEnumerateReferenceSpaces(
    XrSession                                   session,
    uint32_t                                    spaceCapacityInput,
    uint32_t*                                   spaceCountOutput,
    XrReferenceSpaceType*                       spaces);
```

Parameter Descriptions

- `session` is a handle to an XrSession previously created with xrCreateSession.
- `spaceCapacityInput` is the capacity of the spaces array, or 0 to indicate a request to retrieve the required capacity.
- `spaceCountOutput` is a pointer to the count of spaces written, or a pointer to the required capacity in the case that `spaceCapacityInput` is 0.
- `spaces` is a pointer to an application-allocated array that will be filled with the enumerant of each supported reference space. It **can** be `NULL` if `spaceCapacityInput` is 0.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required `spaces` size.

Enumerates the set of reference space types that this runtime supports for a given session. Runtimes **must** always return identical buffer contents from this enumeration for the lifetime of the session.

If a session enumerates support for a given reference space type, calls to xrCreateReferenceSpace **must** succeed for that session, with any transient unavailability of poses expressed later during calls to xrLocateSpace.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- `spaceCountOutput` **must** be a pointer to a `uint32_t` value
- If `spaceCapacityInput` is not `0`, `spaces` **must** be a pointer to an array of `spaceCapacityInput` XrReferenceSpaceType values

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_SIZE_INSUFFICIENT
- XR_ERROR_VALIDATION_FAILURE

7.3.2. xrCreateReferenceSpace

The xrCreateReferenceSpace function is defined as:

```
XrResult xrCreateReferenceSpace(
    XrSession                                   session,
    const XrReferenceSpaceCreateInfo*           createInfo,
    XrSpace*                                    space);
```

Parameter Descriptions

- session is a handle to an XrSession previously created with xrCreateSession.
- createInfo is the XrReferenceSpaceCreateInfo used to specify the space.
- space is the returned space handle.

Creates an XrSpace handle based on a chosen reference space. Application **can** provide an XrPosef to define the position and orientation of the new space's origin within the natural reference frame of the reference space.

Multiple XrSpace handles may exist simultaneously, up to some limit imposed by the runtime. The XrSpace handle **must** be eventually freed via the xrDestroySpace function.

The runtime **must** return XR_ERROR_REFERENCE_SPACE_UNSUPPORTED if the given reference space type is not supported by this `session`.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- `createInfo` **must** be a pointer to a valid XrReferenceSpaceCreateInfo structure
- `space` **must** be a pointer to an XrSpace handle

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_LIMIT_REACHED`
- `XR_ERROR_OUT_OF_MEMORY`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_REFERENCE_SPACE_UNSUPPORTED`
- `XR_ERROR_POSE_INVALID`
- `XR_ERROR_VALIDATION_FAILURE`

The XrReferenceSpaceCreateInfo structure is defined as:

```
typedef struct XrReferenceSpaceCreateInfo {
    XrStructureType         type;
    const void*             next;
    XrReferenceSpaceType    referenceSpaceType;
    XrPosef                 poseInReferenceSpace;
} XrReferenceSpaceCreateInfo;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- referenceSpaceType is the chosen XrReferenceSpaceType.
- poseInReferenceSpace is an XrPosef defining the position and orientation of the new space's origin within the natural reference frame of the reference space.

Valid Usage (Implicit)

- type **must** be XR_TYPE_REFERENCE_SPACE_CREATE_INFO
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- referenceSpaceType **must** be a valid XrReferenceSpaceType value

7.3.3. xrCreateActionSpace

The xrCreateActionSpace function is defined as:

```
XrResult xrCreateActionSpace(
    XrSession                                   session,
    const XrActionSpaceCreateInfo*              createInfo,
    XrSpace*                                    space);
```

Parameter Descriptions

- session is the XrSession to create the action space in.
- createInfo is the XrActionSpaceCreateInfo used to specify the space.
- space is the returned space handle.

Creates an XrSpace handle based on a chosen pose action. Application **can** provide an XrPosef to define the position and orientation of the new space's origin within the natural reference frame of the action space.

Multiple XrSpace handles may exist simultaneously, up to some limit imposed by the runtime. The XrSpace handle must be eventually freed via the xrDestroySpace function or by destroying the parent

XrAction handle.

The runtime **must** return XR_ERROR_ACTION_TYPE_MISMATCH if the action provided in action is not of type XR_ACTION_TYPE_POSE_INPUT.

Valid Usage (Implicit)

- session **must** be a valid XrSession handle
- createInfo **must** be a pointer to a valid XrActionSpaceCreateInfo structure
- space **must** be a pointer to an XrSpace handle

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_ACTION_TYPE_MISMATCH
- XR_ERROR_LIMIT_REACHED
- XR_ERROR_POSE_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_PATH_UNSUPPORTED
- XR_ERROR_PATH_INVALID

The XrActionSpaceCreateInfo structure is defined as:

```
typedef struct XrActionSpaceCreateInfo {
    XrStructureType    type;
    const void*        next;
    XrAction           action;
    XrPath             subactionPath;
    XrPosef            poseInActionSpace;
} XrActionSpaceCreateInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `action` is a handle to a pose XrAction previously created with xrCreateAction.
- `subactionPath` is XR_NULL_PATH or an `XrPath` that was specified when the action was created. If `subactionPath` is a valid path not specified when the action was created the runtime **must** return `XR_ERROR_PATH_UNSUPPORTED`. If this parameter is set, the runtime **must** create a space that is relative to only that subaction's pose binding.
- `poseInActionSpace` is an XrPosef defining the position and orientation of the new space's origin within the natural reference frame of the pose action.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_ACTION_SPACE_CREATE_INFO`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `action` **must** be a valid XrAction handle

7.3.4. xrDestroySpace

The xrDestroySpace function is defined as:

```
XrResult xrDestroySpace(
    XrSpace                                     space);
```

Parameter Descriptions

- `space` is a handle to an XrSpace previously created by a function such as xrCreateReferenceSpace.

XrSpace handles are destroyed using xrDestroySpace. The runtime **may** still use this space if there are active dependencies (e.g, compositions in progress).

Valid Usage (Implicit)

- `space` **must** be a valid XrSpace handle

Thread Safety

- Access to `space`, and any child handles, **must** be externally synchronized

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_HANDLE_INVALID`

7.4. Locating Spaces

Applications use the xrLocateSpace function to find the pose of an XrSpace's origin within a base XrSpace at a given historical or predicted time. If an application wants to know the velocity of the space's origin, it **can** chain an XrSpaceVelocity structure to the `next` pointer of the XrSpaceLocation structure when calling the xrLocateSpace function. Applications **should** inspect the output XrSpaceLocationFlagBits and XrSpaceVelocityFlagBits to determine the validity and tracking status of the components of the location.

7.4.1. xrLocateSpace

xrLocateSpace provides the physical location of a space in a base space at a specified time, if currently known by the runtime.

```
XrResult xrLocateSpace(
    XrSpace                                     space,
    XrSpace                                     baseSpace,
    XrTime                                      time,
    XrSpaceLocation*                            location);
```

Parameter Descriptions

- `space` identifies the target space to locate.
- `baseSpace` identifies the underlying space in which to locate `space`.
- `time` is the time for which the location should be provided.
- `location` provides the location of `space` in `baseSpace`.

For a `time` in the past, the runtime **should** locate the spaces based on the runtime's most accurate current understanding of how the world was at that historical time.

For a `time` in the future, the runtime **should** locate the spaces based on the runtime's most up-to-date prediction of how the world will be at that future time.

The minimum valid range of values for `time` are described in Prediction Time Limits. For values of `time` outside this range, xrLocateSpace **may** return a location with no position and `XR_SPACE_LOCATION_POSITION_VALID_BIT` unset.

Some devices improve their understanding of the world as the device is used. The location returned by xrLocateSpace for a given `space`, `baseSpace` and `time` **may** change over time, even for spaces that track static objects, as one or both spaces adjust their origins.

During tracking loss of `space` relative to `baseSpace`, runtimes **should** continue to provide inferred or last-known `position` and `orientation` values. These inferred poses can, for example, be based on neck model updates, inertial dead reckoning, or a last-known position, so long as it is still reasonable for the application to use that pose. While a runtime is providing position data, it **must** continue to set `XR_SPACE_LOCATION_POSITION_VALID_BIT` but it **can** clear `XR_SPACE_LOCATION_POSITION_TRACKED_BIT` to indicate that the position is inferred or last-known in this way.

If the runtime has not yet observed even a last-known pose for how to locate `space` in `baseSpace` (e.g. one space is an action space bound to a motion controller that has not yet been detected, or the two spaces are in disconnected fragments of the runtime's tracked volume), the runtime **should** return a location with no position and `XR_SPACE_LOCATION_POSITION_VALID_BIT` unset.

The runtime **must** return a location with both `XR_SPACE_LOCATION_POSITION_VALID_BIT` and `XR_SPACE_LOCATION_POSITION_TRACKED_BIT` set when locating `space` and `baseSpace` if both spaces were created relative to the same entity (e.g. two action spaces for the same action), even if the entity is currently untracked. The location in this case is the difference in the two spaces' application-specified transforms relative to that common entity.

The runtime **should** return a location with XR_SPACE_LOCATION_POSITION_VALID_BIT set and XR_SPACE_LOCATION_POSITION_TRACKED_BIT unset for spaces tracking two static entities in the world when their relative pose is known to the runtime. This enables applications to make use of the runtime's latest knowledge of the world, even during tracking loss.

If an XrSpaceVelocity structure is chained to the next pointer of XrSpaceLocation and the velocity is observed or can be calculated by the runtime, the runtime **must** fill in the linear velocity of the origin of space within the reference frame of baseSpace and set the XR_SPACE_VELOCITY_LINEAR_VALID_BIT. Similarly, if an XrSpaceVelocity structure is chained to the next pointer of XrSpaceLocation and the angular velocity is observed or can be calculated by the runtime, the runtime **must** fill in the angular velocity of the origin of space within the reference frame of baseSpace and set the XR_SPACE_VELOCITY_ANGULAR_VALID_BIT.

The following example code shows how an application can get both the location and velocity of a space within a base space using the xrLocateSpace function by chaining an XrSpaceVelocity to the next pointer of XrSpaceLocation and calling xrLocateSpace.

```
XrSpace space;      // previously initialized
XrSpace baseSpace;  // previously initialized
XrTime time;        // previously initialized

XrSpaceVelocity velocity {XR_TYPE_SPACE_VELOCITY};
XrSpaceLocation location {XR_TYPE_SPACE_LOCATION, &velocity};
xrLocateSpace(space, baseSpace, time, &location);
```

Valid Usage (Implicit)

- space **must** be a valid XrSpace handle
- baseSpace **must** be a valid XrSpace handle
- location **must** be a pointer to an XrSpaceLocation structure
- Both of baseSpace and space **must** have been created, allocated, or retrieved from the same XrSession

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_TIME_INVALID

The XrSpaceLocation structure is defined as:

```
typedef struct XrSpaceLocation {
    XrStructureType         type;
    void*                   next;
    XrSpaceLocationFlags    locationFlags;
    XrPosef                 pose;
} XrSpaceLocation;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain, such as XrSpaceVelocity.
- locationFlags is a bitfield, with bit masks defined in XrSpaceLocationFlagBits, to indicate which members contain valid data. If none of the bits are set, no other fields in this structure **should** be considered to be valid or meaningful.
- pose is an XrPosef defining the position and orientation of the origin of xrLocateSpace::space within the reference frame of xrLocateSpace::baseSpace.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_SPACE_LOCATION`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrEyeGazeSampleTimeEXT, XrSpaceVelocity
- `locationFlags` **must** be `0` or a valid combination of XrSpaceLocationFlagBits values

The `locationFlags` member is a bitwise-OR of zero or more of the following flags:

```
// Flag bits for XrSpaceLocationFlags
static const XrSpaceLocationFlags XR_SPACE_LOCATION_ORIENTATION_VALID_BIT = 0x00000001;
static const XrSpaceLocationFlags XR_SPACE_LOCATION_POSITION_VALID_BIT = 0x00000002;
static const XrSpaceLocationFlags XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT = 0x00000004;
static const XrSpaceLocationFlags XR_SPACE_LOCATION_POSITION_TRACKED_BIT = 0x00000008;
```

where the flags have the following meaning:

Flag Descriptions

- XR_SPACE_LOCATION_ORIENTATION_VALID_BIT indicates that the pose field's orientation field contains valid data. For a space location tracking a device with its own inertial tracking, XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT **should** remain set when this bit is set. Applications **must** not read the pose field's orientation if this flag is unset.
- XR_SPACE_LOCATION_POSITION_VALID_BIT indicates that the pose field's position field contains valid data. When a space location loses tracking, runtimes **should** continue to provide valid but untracked position values that are inferred or last-known, so long as it's still meaningful for the application to use that position, clearing XR_SPACE_LOCATION_POSITION_TRACKED_BIT until positional tracking is recovered. Applications **must** not read the pose field's position if this flag is unset.
- XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT indicates that the pose field's orientation field represents an actively tracked orientation. For a space location tracking a device with its own inertial tracking, this bit **should** remain set when XR_SPACE_LOCATION_ORIENTATION_VALID_BIT is set. For a space location tracking an object whose orientation is no longer known during tracking loss (e.g. an observed QR code), runtimes **should** continue to provide valid but untracked orientation values, so long as it's still meaningful for the application to use that orientation.
- XR_SPACE_LOCATION_POSITION_TRACKED_BIT indicates that the pose field's position field represents an actively tracked position. When a space location loses tracking, runtimes **should** continue to provide valid but untracked position values that are inferred or last-known, e.g. based on neck model updates, inertial dead reckoning, or a last-known position, so long as it's still meaningful for the application to use that position.

The XrSpaceVelocity structure is defined as:

```
typedef struct XrSpaceVelocity {
    XrStructureType        type;
    void*                  next;
    XrSpaceVelocityFlags   velocityFlags;
    XrVector3f             linearVelocity;
    XrVector3f             angularVelocity;
} XrSpaceVelocity;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `velocityFlags` is a bitfield, with bit masks defined in XrSpaceVelocityFlagBits, to indicate which members contain valid data. If none of the bits are set, no other fields in this structure **should** be considered to be valid or meaningful.
- `linearVelocity` is the relative linear velocity of the origin of xrLocateSpace::`space` with respect to and expressed in the reference frame of xrLocateSpace::`baseSpace`, in units of meters per second.
- `angularVelocity` is the relative angular velocity of xrLocateSpace::`space` with respect to xrLocateSpace::`baseSpace`. The vector's direction is expressed in the reference frame of xrLocateSpace::`baseSpace` and is parallel to the rotational axis of xrLocateSpace::`space`. The vector's magnitude is the relative angular speed of xrLocateSpace::`space` in radians per second. The vector follows the right-hand rule for torque/rotation.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_SPACE_VELOCITY`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `velocityFlags` **must** be `0` or a valid combination of XrSpaceVelocityFlagBits values

The `velocityFlags` member is a bitwise-OR of zero or more of the following flags:

```
// Flag bits for XrSpaceVelocityFlags
static const XrSpaceVelocityFlags XR_SPACE_VELOCITY_LINEAR_VALID_BIT = 0x00000001;
static const XrSpaceVelocityFlags XR_SPACE_VELOCITY_ANGULAR_VALID_BIT = 0x00000002;
```

where the flags have the following meaning:

Flag Descriptions

- `XR_SPACE_VELOCITY_LINEAR_VALID_BIT` indicates that the `linearVelocity` field contains valid data.
- `XR_SPACE_VELOCITY_ANGULAR_VALID_BIT` indicates that the `angularVelocity` field contains valid data.

Chapter 8. View Configurations

A **view configuration** is a semantically meaningful set of one or more views for which an application can render images. A **primary view configuration** is a view configuration intended to be presented to the viewer interacting with the XR application. This distinction allows the later addition of additional views, for example views which are intended for spectators.

A typical head-mounted VR system has a view configuration with two views, while a typical phone-based AR system has a view configuration with a single view. A simple multi-wall projection-based (CAVE-like) VR system may have a view configuration with at least one view for each display surface (wall, floor, ceiling) in the room.

For any supported form factor, a system will support one or more primary view configurations. Supporting more than one primary view configuration can be useful if a system supports a special view configuration optimized for the hardware but also supports a more broadly used view configuration as a compatibility fallback.

View configurations are identified with an XrViewConfigurationType.

8.1. Primary View Configurations

```
typedef enum XrViewConfigurationType {
    XR_VIEW_CONFIGURATION_TYPE_PRIMARY_MONO = 1,
    XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO = 2,
    XR_VIEW_CONFIGURATION_TYPE_PRIMARY_QUAD_VARJO = 1000037000,
    XR_VIEW_CONFIGURATION_TYPE_SECONDARY_MONO_FIRST_PERSON_OBSERVER_MSFT = 1000054000,
    XR_VIEW_CONFIGURATION_TYPE_MAX_ENUM = 0x7FFFFFFF
} XrViewConfigurationType;
```

The application selects its primary view configuration type when calling xrBeginSession, and that configuration remains constant for the lifetime of the session, until xrEndSession is called.

The number of views and the semantic meaning of each view index within a given view configuration is well-defined, specified below for all core view configurations. The predefined primary view configuration types are:

Enumerant Descriptions

- XR_VIEW_CONFIGURATION_TYPE_PRIMARY_MONO. One view representing the form factor's one primary display. For example, an AR phone's screen. This configuration requires one element in XrViewConfigurationProperties and one projection in each XrCompositionLayerProjection layer.
- XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO. Two views representing the form factor's two primary displays, which map to a left-eye and right-eye view. This configuration requires two views in XrViewConfigurationProperties and two views in each XrCompositionLayerProjection layer. View index 0 **must** represent the left eye and view index 1 **must** represent the right eye.

8.2. View Configuration API

First an application needs to select which primary view configuration it wants to use. If it supports multiple configurations, an application **can** call xrEnumerateViewConfigurations before creating an XrSession to get a list of the view configuration types supported for a given system.

The application **can** then call xrGetViewConfigurationProperties and xrEnumerateViewConfigurationViews to get detailed information about each view configuration type and its individual views.

8.2.1. xrEnumerateViewConfigurations

The xrEnumerateViewConfigurations function is defined as:

```
XrResult xrEnumerateViewConfigurations(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    uint32_t                                    viewConfigurationTypeCapacityInput,
    uint32_t*                                   viewConfigurationTypeCountOutput,
    XrViewConfigurationType*                    viewConfigurationTypes);
```

Parameter Descriptions

- `instance` is the instance from which `systemId` was retrieved.
- `systemId` is the `XrSystemId` whose view configurations will be enumerated.
- `viewConfigurationsTypeCapacityInput` is the capacity of the `viewConfigurations` array, or 0 to indicate a request to retrieve the required capacity.
- `viewConfigurationsTypeCountOutput` is a pointer to the count of `viewConfigurations` written, or a pointer to the required capacity in the case that `viewConfigurationsTypeCapacityInput` is 0.
- `viewConfigurationsTypes` is a pointer to an array of XrViewConfigurationType values, but **can** be `NULL` if `viewConfigurationsTypeCapacityInput` is 0.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required `viewConfigurations` size.

xrEnumerateViewConfigurations enumerates the view configuration types supported by the `XrSystemId`. The supported set for that system **must** not change during the lifetime of its XrInstance. The returned list of primary view configurations **should** be in order from what the runtime considered highest to lowest user preference. Thus the first enumerated view configuration type **should** be the one the runtime prefers the application to use if possible.

Runtimes **must** always return identical buffer contents from this enumeration for the given `systemId` and for the lifetime of the instance.

Valid Usage (Implicit)

- `instance` **must** be a valid XrInstance handle
- `viewConfigurationTypeCountOutput` **must** be a pointer to a `uint32_t` value
- If `viewConfigurationTypeCapacityInput` is not `0`, `viewConfigurationTypes` **must** be a pointer to an array of `viewConfigurationTypeCapacityInput` XrViewConfigurationType values

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_SYSTEM_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_SIZE_INSUFFICIENT

8.2.2. xrGetViewConfigurationProperties

The xrGetViewConfigurationProperties function is defined as:

```
XrResult xrGetViewConfigurationProperties(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    XrViewConfigurationType                     viewConfigurationType,
    XrViewConfigurationProperties*              configurationProperties);
```

Parameter Descriptions

- `instance` is the instance from which `systemId` was retrieved.
- `systemId` is the `XrSystemId` whose view configuration is being queried.
- `viewConfigurationType` is the XrViewConfigurationType of the configuration to get.
- `configurationProperties` is a pointer to view configuration properties to return.

xrGetViewConfigurationProperties queries properties of an individual view configuration. Applications **must** use one of the supported view configuration types returned by xrEnumerateViewConfigurations. If `viewConfigurationType` is not supported by this XrInstance the runtime **must** return `XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED`.

Valid Usage (Implicit)

- `instance` **must** be a valid XrInstance handle
- `viewConfigurationType` **must** be a valid XrViewConfigurationType value
- `configurationProperties` **must** be a pointer to an XrViewConfigurationProperties structure

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_SYSTEM_INVALID`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED`

8.2.3. XrViewConfigurationProperties

The XrViewConfigurationProperties structure is defined as:

```
typedef struct XrViewConfigurationProperties {
    XrStructureType            type;
    void*                      next;
    XrViewConfigurationType    viewConfigurationType;
    XrBool32                   fovMutable;
} XrViewConfigurationProperties;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `viewConfigurationType` is the XrViewConfigurationType of the configuration.
- `fovMutable` indicates if the view field of view can be modified by the application.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_VIEW_CONFIGURATION_PROPERTIES`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `viewConfigurationType` **must** be a valid XrViewConfigurationType value

8.2.4. xrEnumerateViewConfigurationViews

The xrEnumerateViewConfigurationViews function is defined as:

```
XrResult xrEnumerateViewConfigurationViews(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    XrViewConfigurationType                     viewConfigurationType,
    uint32_t                                    viewCapacityInput,
    uint32_t*                                   viewCountOutput,
    XrViewConfigurationView*                    views);
```

Parameter Descriptions

- `instance` is the instance from which `systemId` was retrieved.
- `systemId` is the `XrSystemId` whose view configuration is being queried.
- `viewConfigurationType` is the XrViewConfigurationType of the configuration to get.
- `viewCapacityInput` is the capacity of the `views` array, or 0 to indicate a request to retrieve the required capacity.
- `viewCountOutput` is a pointer to the count of `views` written, or a pointer to the required capacity in the case that `viewCapacityInput` is 0.
- `views` is a pointer to an array of XrViewConfigurationView values, but **can** be `NULL` if `viewCapacityInput` is 0.

Each XrViewConfigurationType defines the number of views associated with it. Applications can query more details of each view element using xrEnumerateViewConfigurationViews. If the supplied `viewConfigurationType` is not supported by this XrInstance and `XrSystemId`, the runtime **must** return `XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED`.

Runtimes **must** always return identical buffer contents from this enumeration for the given `systemId` and `viewConfigurationType` for the lifetime of the instance.

Valid Usage (Implicit)

- `instance` **must** be a valid XrInstance handle
- `viewConfigurationType` **must** be a valid XrViewConfigurationType value
- `viewCountOutput` **must** be a pointer to a `uint32_t` value
- If `viewCapacityInput` is not `0`, `views` **must** be a pointer to an array of `viewCapacityInput` XrViewConfigurationView structures

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_SYSTEM_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED
- XR_ERROR_SIZE_INSUFFICIENT

8.2.5. XrViewConfigurationView

Each XrViewConfigurationView specifies properties related to rendering of an individual view within a view configuration.

The XrViewConfigurationView structure is defined as:

```
typedef struct XrViewConfigurationView {
    XrStructureType    type;
    void*              next;
    uint32_t           recommendedImageRectWidth;
    uint32_t           maxImageRectWidth;
    uint32_t           recommendedImageRectHeight;
    uint32_t           maxImageRectHeight;
    uint32_t           recommendedSwapchainSampleCount;
    uint32_t           maxSwapchainSampleCount;
} XrViewConfigurationView;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `recommendedImageRectWidth` is the optimal width of `imageRect` to use when rendering this view into a swapchain.
- `maxImageRectWidth` is the maximum width of `imageRect` supported when rendering this view into a swapchain.
- `recommendedImageRectHeight` is the optimal height of `imageRect` to use when rendering this view into a swapchain.
- `maxImageRectHeight` is the maximum height of `imageRect` supported when rendering this view into a swapchain.
- `recommendedSwapchainSampleCount` is the recommended number of sub-data element samples to create for each swapchain image that will be rendered into for this view.
- `maxSwapchainSampleCount` is the maximum number of sub-data element samples supported for swapchain images that will be rendered into for this view.

See XrSwapchainSubImage for more information about `imageRect` values, and XrSwapchainCreateInfo for more information about creating swapchains appropriately sized to support those `imageRect` values.

The array of XrViewConfigurationView returned by the runtime **must** adhere to the rules defined in XrViewConfigurationType, such as the count and association to the left and right eyes.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_VIEW_CONFIGURATION_VIEW`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrFoveatedViewConfigurationViewVARJO, XrViewConfigurationDepthRangeEXT, XrViewConfigurationViewFovEPIC

8.3. Example View Configuration Code

```
XrInstance instance; // previously initialized
XrSystemId system;   // previously initialized
XrSession session;   // previously initialized
XrSpace sceneSpace;  // previously initialized

// Enumerate the view configurations paths.
```

```
uint32_t configurationCount;
CHK_XR(xrEnumerateViewConfigurations(instance, system, 0, &configurationCount, nullptr));

std::vector<XrViewConfigurationType> configurationTypes(configurationCount);
CHK_XR(xrEnumerateViewConfigurations(instance, system, configurationCount,
&configurationCount, configurationTypes.data()));

bool configFound = false;
for(uint32_t i = 0; i < configurationCount; ++i)
{
    if (configurationTypes[i] == XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO)
    {
        configFound = true;
        break;  // Pick the first supported, i.e. preferred, view configuration.
    }
}

if (!configFound)
    return;   // Cannot support any view configuration of this system.

// Get detailed information of each view element.
uint32_t viewCount;
CHK_XR(xrEnumerateViewConfigurationViews(instance, system,
    XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO,
    0,
    &viewCount,
    nullptr));

std::vector<XrViewConfigurationView> configViews(viewCount,
{XR_TYPE_VIEW_CONFIGURATION_VIEW});
CHK_XR(xrEnumerateViewConfigurationViews(instance, system,
    XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO,
    viewCount,
    &viewCount,
    configViews.data()));

// Set the primary view configuration for the session.
XrSessionBeginInfo beginInfo = {XR_TYPE_SESSION_BEGIN_INFO};
beginInfo.primaryViewConfigurationType = XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO;
CHK_XR(xrBeginSession(session, &beginInfo));

// Allocate a buffer according to viewCount.
std::vector<XrView> views(viewCount, {XR_TYPE_VIEW});

// Run a per-frame loop.
while (!quit)
{
    // Wait for a new frame.
```

```
    XrFrameWaitInfo frameWaitInfo{XR_TYPE_FRAME_WAIT_INFO};
    XrFrameState frameState{XR_TYPE_FRAME_STATE};
    CHK_XR(xrWaitFrame(session, &frameWaitInfo, &frameState));

    // Begin frame immediately before GPU work
    XrFrameBeginInfo frameBeginInfo { XR_TYPE_FRAME_BEGIN_INFO };
    CHK_XR(xrBeginFrame(session, &frameBeginInfo));

    std::vector<XrCompositionLayerBaseHeader*> layers;
    XrCompositionLayerProjectionView projViews[2] = { /*...*/ };
    XrCompositionLayerProjection layerProj{ XR_TYPE_COMPOSITION_LAYER_PROJECTION};

    if (frameState.shouldRender) {
        XrViewLocateInfo viewLocateInfo{XR_TYPE_VIEW_LOCATE_INFO};
        viewLocateInfo.displayTime = frameState.predictedDisplayTime;
        viewLocateInfo.space = sceneSpace;

        XrViewState viewState{XR_TYPE_VIEW_STATE};
        XrView views[2] = { {XR_TYPE_VIEW}, {XR_TYPE_VIEW}};
        uint32_t viewCountOutput;
        CHK_XR(xrLocateViews(session, &viewLocateInfo, &viewState, configViews.size(),
&viewCountOutput, views));

        // ...
        // Use viewState and frameState for scene render, and fill in projViews[2]
        // ...

        // Assemble composition layers structure
        layerProj.layerFlags = XR_COMPOSITION_LAYER_BLEND_TEXTURE_SOURCE_ALPHA_BIT;
        layerProj.space = sceneSpace;
        layerProj.viewCount = 2;
        layerProj.views = projViews;
        layers.push_back(reinterpret_cast<XrCompositionLayerBaseHeader*>(&layerProj));
    }

    // End frame and submit layers, even if layers is empty due to shouldRender = false
    XrFrameEndInfo frameEndInfo{ XR_TYPE_FRAME_END_INFO};
    frameEndInfo.displayTime = frameState.predictedDisplayTime;
    frameEndInfo.environmentBlendMode = XR_ENVIRONMENT_BLEND_MODE_OPAQUE;
    frameEndInfo.layerCount = (uint32_t)layers.size();
    frameEndInfo.layers = layers.data();
    CHK_XR(xrEndFrame(session, &frameEndInfo));
}
```

Chapter 9. Session

```
XR_DEFINE_HANDLE(XrSession)
```

A session represents an application's intention to display XR content to the user.

First, the application creates a session by choosing a system and a graphics API and calling xrCreateSession, which creates a session in the `XR_SESSION_STATE_IDLE` state. The application then sets up an xrPollEvent loop to monitor for session state changes delivered through the XrEventDataSessionStateChanged event. When the runtime determines that the system is ready to start transitioning to this session's XR content, it notifies the application that its session has moved into the `XR_SESSION_STATE_READY` state. When the application is ready to proceed and display its XR content, it calls xrBeginSession and starts its frame loop, which starts its session running. While the session is running, the application is expected to continuously run its frame loop by calling xrWaitFrame, xrBeginFrame and xrEndFrame each frame, to establish synchronization with the runtime. Once the runtime is synchronized with the application's frame loop and ready to display its frames, the session will move into the `XR_SESSION_STATE_SYNCHRONIZED` state. For frames where xrWaitFrame returns an XrFrameState with `shouldRender` set to true, the application should render its composition layers and submit them to xrEndFrame. If the application desires to leave a running session, it should call the xrRequestExitSession function to request that the runtime transition its session to the `XR_SESSION_STATE_STOPPING` state as soon as possible. Once the application reaches the `XR_SESSION_STATE_STOPPING` state, it can call xrEndSession to stop the XR session, after which the session will transition through `XR_SESSION_STATE_IDLE` to the `XR_SESSION_STATE_EXITING` state.

On platforms that have applications with lifecycle managed by the system, session state changes may be implicitly triggered by application lifecycle state changes. On such platforms, using platform-specific methods to alter application lifecycle state may be the preferred method of provoking session state changes. The behavior of xrRequestExitSession is not altered, however explicit session exit **may** not interact with the platform-specific application lifecycle.

A session is considered **running** after a successful call to xrBeginSession and remains running until any call is made to xrEndSession. Certain functions are only valid to call when a session is running, such as xrWaitFrame, or else the `XR_ERROR_SESSION_NOT_RUNNING` error **must** be returned by the runtime.

A session is considered **not running** before a successful call to xrBeginSession and becomes not running again after any call is made to xrEndSession. Certain functions are only valid to call when a session is not running, such as xrBeginSession, or else the `XR_ERROR_SESSION_RUNNING` error **must** be returned by the runtime.

If an error is returned from xrBeginSession, the session remains in its current running or not running state. Calling xrEndSession always transitions a session to the not running state, regardless of any errors returned.

Only running sessions may become focused sessions that receive XR input. When a session is not running, the application **must** not submit frames. This is important because without a running session, the runtime no longer has to spend resources on sub-systems (tracking etc.) that are no longer needed by the application.

9.1. Session Lifecycle

To present graphical content on an output device, OpenXR applications need to pick a graphics API which is supported by the runtime. Unextended OpenXR does not support any graphics APIs natively but provides a number of extensions of which each runtime can support any subset. These extensions can be activated during XrInstance create time.

During XrSession creation the application **must** provide information about which graphics API it intends to use by adding an `XrGraphicsBinding*` struct of one (and only one) of the enabled graphics API extensions to the next chain of XrSessionCreateInfo. The application **must** call the `xrGet*GraphicsRequirements` method (where `*` is a placeholder) provided by the chosen graphics API extension before attempting to create the session (for example xrGetD3D11GraphicsRequirementsKHR, xrGetD3D12GraphicsRequirementsKHR, xrGetOpenGLGraphicsRequirementsKHR, xrGetOpenGLESGraphicsRequirementsKHR, xrGetVulkanGraphicsRequirementsKHR).

Unless specified differently in the graphics API extension, the application is responsible for creating a valid graphics device binding based on the requirements returned by `xrGet*GraphicsRequirements` methods (for details refer to the extension specification of the graphics API).

The xrCreateSession function is defined as:

```
XrResult xrCreateSession(
    XrInstance                                  instance,
    const XrSessionCreateInfo*                  createInfo,
    XrSession*                                  session);
```

Parameter Descriptions

- `instance` is the instance from which `systemId` was retrieved.
- `createInfo` is a pointer to an XrSessionCreateInfo structure containing information about how to create the session.
- `session` is a pointer to a handle in which the created XrSession is returned.

Creates a session using the provided `createInfo` and returns a handle to that session. This session is created in the `XR_SESSION_STATE_IDLE` state, and a corresponding XrEventDataSessionStateChanged event to the `XR_SESSION_STATE_IDLE` state **must** be generated as the first such event for the new session.

The runtime **must** return XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING (XR_ERROR_VALIDATION_FAILURE may be returned due to legacy behavior) on calls to xrCreateSession if a function named like xrGet*GraphicsRequirements has not been called for the same instance and XrSessionCreateInfo ::systemId. (See graphics binding extensions for details.)

Valid Usage (Implicit)

- instance **must** be a valid XrInstance handle
- createInfo **must** be a pointer to a valid XrSessionCreateInfo structure
- session **must** be a pointer to an XrSession handle

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_LIMIT_REACHED
- XR_ERROR_INITIALIZATION_FAILED
- XR_ERROR_SYSTEM_INVALID
- XR_ERROR_GRAPHICS_DEVICE_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING

The XrSessionCreateInfo structure is defined as:

```
typedef struct XrSessionCreateInfo {
    XrStructureType         type;
    const void*             next;
    XrSessionCreateFlags    createFlags;
    XrSystemId              systemId;
} XrSessionCreateInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR. Note that in most cases one graphics API extension specific struct needs to be in this next chain.
- `createFlags` identifies XrSessionCreateFlags that apply to the creation.
- `systemId` is the `XrSystemId` representing the system of devices to be used by this session.

Valid Usage

- `systemId` **must** be a valid `XrSystemId` or `XR_ERROR_SYSTEM_INVALID` **must** be returned.
- `next`, unless otherwise specified via an extension, **must** contain exactly one graphics API binding structure (a structure whose name begins with "XrGraphicsBinding") or `XR_ERROR_GRAPHICS_DEVICE_INVALID` **must** be returned.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_SESSION_CREATE_INFO`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrGraphicsBindingD3D11KHR, XrGraphicsBindingD3D12KHR, XrGraphicsBindingEGLMNDX, XrGraphicsBindingOpenGLESAndroidKHR, XrGraphicsBindingOpenGLWaylandKHR, XrGraphicsBindingOpenGLWin32KHR, XrGraphicsBindingOpenGLXcbKHR, XrGraphicsBindingOpenGLXlibKHR, XrGraphicsBindingVulkanKHR, XrHolographicWindowAttachmentMSFT, XrSessionCreateInfoOverlayEXTX
- `createFlags` **must** be `0`

The XrSessionCreateFlags include:

```
// Flag bits for XrSessionCreateFlags
```

There are currently no session creation flags. This is reserved for future use.

The xrDestroySession function is defined as.

```
XrResult xrDestroySession(
    XrSession                                   session);
```

Parameter Descriptions

- `session` is the session to destroy.

XrSession handles are destroyed using xrDestroySession. When an XrSession is destroyed, all handles that are children of that XrSession are also destroyed.

The application is responsible for ensuring that it has no calls using `session` in progress when the session is destroyed.

xrDestroySession can be called when the session is in any session state.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle

Thread Safety

- Access to `session`, and any child handles, **must** be externally synchronized

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_HANDLE_INVALID`

9.2. Session Control

The xrBeginSession function is defined as:

```
XrResult xrBeginSession(
    XrSession                                   session,
    const XrSessionBeginInfo*                   beginInfo);
```

Parameter Descriptions

- `session` is a valid XrSession handle.
- `beginInfo` is a pointer to an XrSessionBeginInfo structure.

When the application receives XrEventDataSessionStateChanged event with the `XR_SESSION_STATE_READY` state, the application **should** then call xrBeginSession to start rendering frames for display to the user.

After this function successfully returns, the session is considered to be running. The application **should** then start its frame loop consisting of some sequence of xrWaitFrame/xrBeginFrame /xrEndFrame calls.

If the session is already running when the application calls xrBeginSession, the runtime **must** return error `XR_ERROR_SESSION_RUNNING`. If the session is not running when the application calls xrBeginSession, but the session is not yet in the `XR_SESSION_STATE_READY` state, the runtime **must** return error `XR_ERROR_SESSION_NOT_READY`.

Note that a runtime **may** decide not to show the user any given frame from a session at any time, for example if the user has switched to a different application's running session. The application should check whether xrWaitFrame returns an XrFrameState with `shouldRender` set to true before rendering a given frame to determine whether that frame will be visible to the user.

Runtime session frame state **must** start in a reset state when a session transitions to running so that no state is carried over from when the same session was previously running.

If `primaryViewConfigurationType` in `beginInfo` is not supported by the `XrSystemId` used to create the `session`, the runtime **must** return `XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED`.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- `beginInfo` **must** be a pointer to a valid XrSessionBeginInfo structure

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_SESSION_NOT_READY
- XR_ERROR_SESSION_RUNNING
- XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED

The XrSessionBeginInfo structure is defined as:

```
typedef struct XrSessionBeginInfo {
    XrStructureType            type;
    const void*                next;
    XrViewConfigurationType    primaryViewConfigurationType;
} XrSessionBeginInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `primaryViewConfigurationType` is the XrViewConfigurationType to use during this session to provide images for the form factor's primary displays.

Valid Usage (Implicit)

- type **must** be XR_TYPE_SESSION_BEGIN_INFO
- next **must** be NULL or a valid pointer to the next structure in a structure chain. See also: XrSecondaryViewConfigurationSessionBeginInfoMSFT
- primaryViewConfigurationType **must** be a valid XrViewConfigurationType value

The xrEndSession function is defined as:

```
XrResult xrEndSession(
    XrSession                                   session);
```

Parameter Descriptions

- session is a handle to a running XrSession.

When the application receives XrEventDataSessionStateChanged event with the XR_SESSION_STATE_STOPPING state, the application should stop its frame loop and then call xrEndSession to end the running session. This function signals to the runtime that the application will no longer call xrWaitFrame, xrBeginFrame or xrEndFrame from any thread allowing the runtime to safely transition the session to XR_SESSION_STATE_IDLE. The application **must** also avoid reading input state or sending haptic output after calling xrEndSession.

If the session is not running when the application calls xrEndSession, the runtime **must** return error XR_ERROR_SESSION_NOT_RUNNING. If the session is still running when the application calls xrEndSession, but the session is not yet in the XR_SESSION_STATE_STOPPING state, the runtime **must** return error XR_ERROR_SESSION_NOT_STOPPING.

If the application wishes to exit a running session, the application can call xrRequestExitSession so that the session transitions from XR_SESSION_STATE_IDLE to XR_SESSION_STATE_EXITING.

Valid Usage (Implicit)

- session **must** be a valid XrSession handle

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_SESSION_NOT_STOPPING
- XR_ERROR_SESSION_NOT_RUNNING
- XR_ERROR_VALIDATION_FAILURE

The xrRequestExitSession function is defined as:

```
XrResult xrRequestExitSession(
    XrSession                                   session);
```

Parameter Descriptions

- session is a handle to a running XrSession.

An application can only call xrEndSession when the session is in the XR_SESSION_STATE_STOPPING state, which allows runtimes to seamlessly transition from one application's session to another. When an application wishes to exit a running session, the application can call xrRequestExitSession, requesting that the runtime transition through the various intermediate session states including XR_SESSION_STATE_STOPPING to XR_SESSION_STATE_EXITING.

If session is not running when xrRequestExitSession is called, XR_ERROR_SESSION_NOT_RUNNING **must** be returned.

Valid Usage (Implicit)

- session **must** be a valid XrSession handle

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_SESSION_NOT_RUNNING
- XR_ERROR_VALIDATION_FAILURE

9.3. Session States

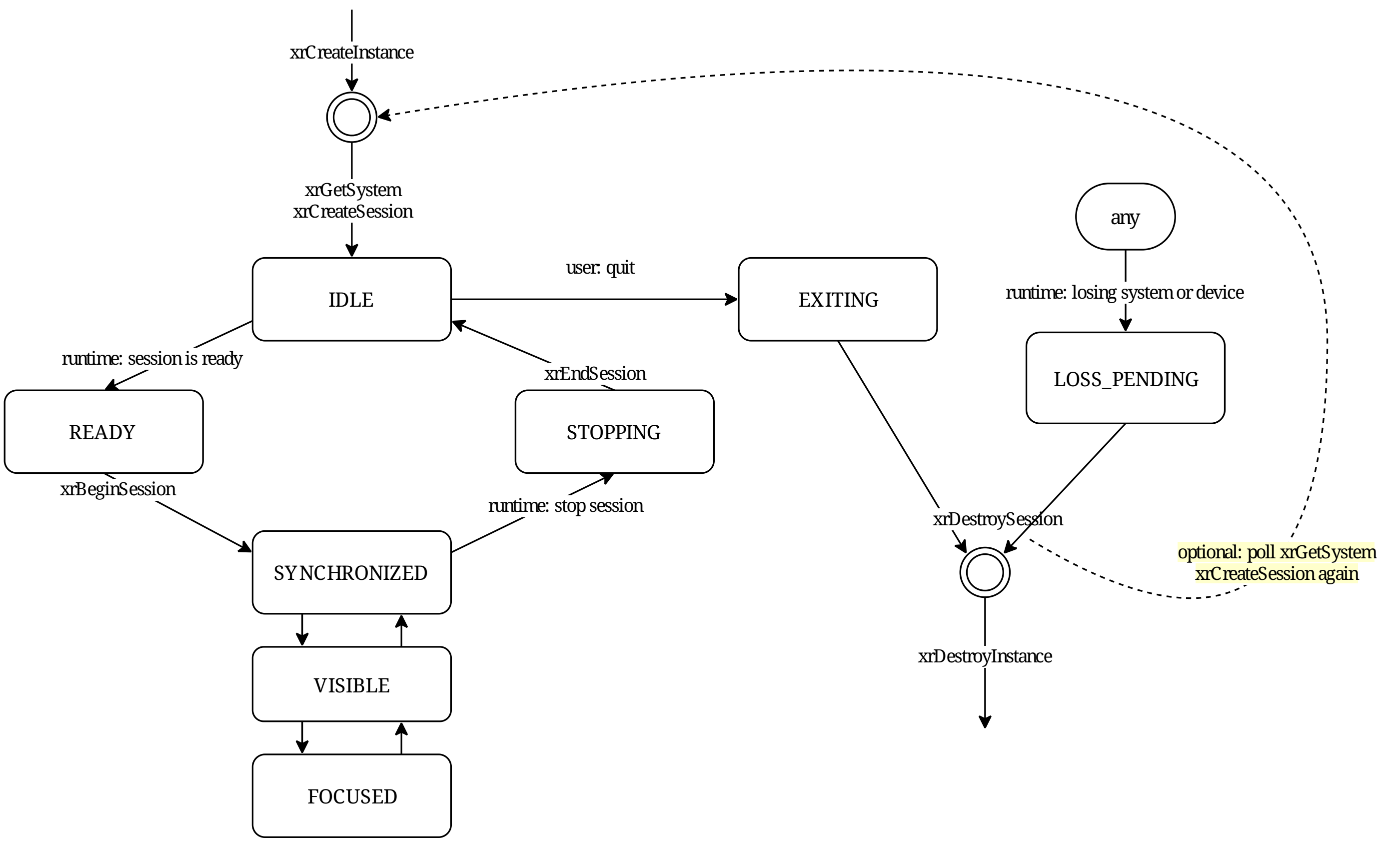


Figure 2. Session Life-cycle

While events can be expanded upon, there are a minimum set of lifecycle events which can occur which all OpenXR applications must be aware of. These events are detailed below.

9.3.1. XrEventDataSessionStateChanged

The XrEventDataSessionStateChanged structure is defined as:

```
typedef struct XrEventDataSessionStateChanged {
     XrStructureType    type;
    const void*         next;
    XrSession           session;
    XrSessionState      state;
    XrTime              time;
} XrEventDataSessionStateChanged;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `session` is the XrSession which has changed state.
- `state` is the current XrSessionState of the `session`.
- `time` is an `XrTime` which indicates the time of the state change.

Receiving the XrEventDataSessionStateChanged event structure indicates that the application has changed lifecycle state.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_EVENT_DATA_SESSION_STATE_CHANGED`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `session` **must** be a valid XrSession handle
- `state` **must** be a valid XrSessionState value

The XrSessionState enumerates the possible session lifecycle states:

```
typedef enum XrSessionState {
    XR_SESSION_STATE_UNKNOWN = 0,
    XR_SESSION_STATE_IDLE = 1,
    XR_SESSION_STATE_READY = 2,
    XR_SESSION_STATE_SYNCHRONIZED = 3,
    XR_SESSION_STATE_VISIBLE = 4,
    XR_SESSION_STATE_FOCUSED = 5,
    XR_SESSION_STATE_STOPPING = 6,
    XR_SESSION_STATE_LOSS_PENDING = 7,
    XR_SESSION_STATE_EXITING = 8,
    XR_SESSION_STATE_MAX_ENUM = 0x7FFFFFFF
} XrSessionState;
```

Enumerant Descriptions

- XR_SESSION_STATE_UNKNOWN. An unknown state. The runtime **must** not return this value in an XrEventDataSessionStateChanged event.
- XR_SESSION_STATE_IDLE. The initial state after calling xrCreateSession or returned to after calling xrEndSession.
- XR_SESSION_STATE_READY. The application is ready to call xrBeginSession and sync its frame loop with the runtime.
- XR_SESSION_STATE_SYNCHRONIZED. The application has synced its frame loop with the runtime but is not visible to the user.
- XR_SESSION_STATE_VISIBLE. The application has synced its frame loop with the runtime and is visible to the user but cannot receive XR input.
- XR_SESSION_STATE_FOCUSED. The application has synced its frame loop with the runtime, is visible to the user and can receive XR input.
- XR_SESSION_STATE_STOPPING. The application should exit its frame loop and call xrEndSession.
- XR_SESSION_STATE_LOSS_PENDING. The session is in the process of being lost. The application should destroy the current session and can optionally recreate it.
- XR_SESSION_STATE_EXITING. The application should end its XR experience and not automatically restart it.

The XR_SESSION_STATE_UNKNOWN state **must** not be returned by the runtime, and is only defined to avoid 0 being a valid state.

Receiving the XR_SESSION_STATE_IDLE state indicates that the runtime considers the session is idle. Applications in this state **should** minimize resource consumption but continue to call xrPollEvent at some reasonable cadence.

Receiving the XR_SESSION_STATE_READY state indicates that the runtime desires the application to prepare rendering resources, begin its session and synchronize its frame loop with the runtime. The application does this by successfully calling xrBeginSession and then running its frame loop by calling xrWaitFrame, xrBeginFrame and xrEndFrame in a loop. If the runtime wishes to return the session to the XR_SESSION_STATE_IDLE state, it **must** wait until the application calls xrBeginSession. After returning from the xrBeginSession call, the runtime may then immediately transition forward through the XR_SESSION_STATE_SYNCHRONIZED state to the XR_SESSION_STATE_STOPPING state, to request that the application end this session. If the system supports a user engagement sensor and runtime is in XR_SESSION_STATE_IDLE state, the runtime **should** not transition to the XR_SESSION_STATE_READY state until the user starts engaging with the device.

Receiving the XR_SESSION_STATE_SYNCHRONIZED state indicates that the application has synchronized its frame loop with the runtime, but its frames are not visible to the user. The application **should** continue running its frame loop by calling xrWaitFrame, xrBeginFrame and xrEndFrame, although it should avoid heavy GPU work so that other visible applications can take CPU and GPU precedence. The application can save resources here by skipping rendering and not submitting any composition layers until xrWaitFrame returns an XrFrameState with shouldRender set to true. A runtime **may** use this frame synchronization to facilitate seamless switching from a previous XR application to this application on a frame boundary.

Receiving the XR_SESSION_STATE_VISIBLE state indicates that the application has synchronized its frame loop with the runtime, and the session's frames will be visible to the user, but the session is not eligible to receive XR input. An application may be visible but not have focus, for example when the runtime is composing a modal pop-up on top of the application's rendered frames. The application **should** continue running its frame loop, rendering and submitting its composition layers, although it may wish to pause its experience, as users cannot interact with the application at this time. It is important for applications to continue rendering when visible, even when they do not have focus, so the user continues to see something reasonable underneath modal pop-ups. Runtimes **should** make input actions inactive while the application is unfocused, and applications should react to an inactive input action by skipping rendering of that action's input avatar (depictions of hands or other tracked objects controlled by the user).

Receiving the XR_SESSION_STATE_FOCUSED state indicates that the application has synchronized its frame loop with the runtime, the session's frames will be visible to the user, and the session is eligible to receive XR input. The runtime **should** only give one session XR input focus at any given time. The application **should** be running its frame loop, rendering and submitting composition layers, including input avatars (depictions of hands or other tracked objects controlled by the user) for any input actions that are active. The runtime **should** avoid rendering its own input avatars when an application is focused, unless input from a given source is being captured by the runtime at the moment.

Receiving the XR_SESSION_STATE_STOPPING state indicates that the runtime has determined that the application should halt its rendering loop. Applications **should** exit their rendering loop and call xrEndSession when in this state. A possible reason for this would be to minimize contention between multiple applications. If the system supports a user engagement sensor and the session is running, the runtime **should** transition to the XR_SESSION_STATE_STOPPING state when the user stops engaging with the device.

Receiving the XR_SESSION_STATE_EXITING state indicates the runtime wishes the application to terminate its XR experience, typically due to a user request via a runtime user interface. Applications **should** gracefully end their process when in this state if they do not have a non-XR user experience.

Receiving the XR_SESSION_STATE_LOSS_PENDING state indicates the runtime is no longer able to operate with the current session, for example due to the loss of a display hardware connection. An application **should** call xrDestroySession and **may** end its process or decide to poll xrGetSystem at some reasonable cadence to get a new XrSystemId, and re-initialize all graphics resources related to the new system, and then create a new session using xrCreateSession. After the event is queued, subsequent calls to functions that accept XrSession parameters **must** no longer return any success code other than XR_SESSION_LOSS_PENDING for the given XrSession handle. The XR_SESSION_LOSS_PENDING success result is returned for an unspecified grace period of time, and the functions that return it simulate success in their behavior. If the runtime has no reasonable way to successfully complete a given function (e.g. xrCreateSwapchain) when a lost session is pending, or if the runtime is not able to provide the application a grace period, the runtime **may** return XR_ERROR_SESSION_LOST. Thereafter, functions which accept XrSession parameters for the lost session **may** return XR_ERROR_SESSION_LOST to indicate that the function failed and the given session was lost. The XrSession handle and child handles are henceforth unusable and **should** be destroyed by the application in order to immediately free up resources associated with those handles.

Chapter 10. Rendering

10.1. Swapchain Image Management

```
XR_DEFINE_HANDLE(XrSwapchain)
```

Normal XR applications will want to present rendered images to the user. To allow this, the runtime provides images organized in swapchains for the application to render into. The runtime **must** allow applications to create multiple swapchains.

Swapchain image format support by the runtime is specified by the xrEnumerateSwapchainFormats function. Runtimes **should** support `R8G8B8A8` and `R8G8B8A8 sRGB` formats if possible.

Swapchain images **can** be 2D or 2D Array.

Rendering operations involving composition of submitted layers should be assumed to be internally performed by the runtime in linear color space. Images submitted in sRGB color space must be created using an API-specific sRGB format (e.g. `DXGI_FORMAT_R8G8B8A8_UNORM_SRGB`, `GL_SRGB8_ALPHA8`, `VK_FORMAT_R8G8B8A8_SRGB`) to apply automatic sRGB-to-linear conversion when read by the runtime. All other formats will be treated as linear values.

> *Note*
>
> OpenXR applications should avoid submitting linear encoded 8 bit color data (e.g. `DXGI_FORMAT_R8G8B8A8_UNORM`) whenever possible as it may result in color banding.
>
> Gritz, L. and d'Eon, E. 2007. The Importance of Being Linear. In: H. Nguyen, ed., *GPU Gems 3*. Addison-Wesley Professional. https://developer.nvidia.com/gpugems/gpugems3/part-iv-image-effects/chapter-24-importance-being-linear

> *Note*
>
> DXGI resources will be created with their associated TYPELESS format, but the runtime will use the application-specified format for reading the data.

The xrEnumerateSwapchainFormats function is defined as:

```
XrResult xrEnumerateSwapchainFormats(
    XrSession                                   session,
    uint32_t                                    formatCapacityInput,
    uint32_t*                                   formatCountOutput,
    int64_t*                                    formats);
```

Parameter Descriptions

- `session` is the session that enumerates the supported formats.
- `formatCapacityInput` is the capacity of the `formats`, or 0 to retrieve the required capacity.
- `formatCountOutput` is a pointer to the count of `uint64_t` formats written, or a pointer to the required capacity in the case that `formatCapacityInput` is `0`.
- `formats` is a pointer to an array of `int64_t` format ids, but **can** be `NULL` if `formatCapacityInput` is `0`. The format ids are specific to the specified graphics API.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required `formats` size.

xrEnumerateSwapchainFormats enumerates the texture formats supported by the current session. The type of formats returned are dependent on the graphics API specified in xrCreateSession. For example, if a DirectX graphics API was specified, then the enumerated formats correspond to the DXGI formats, such as `DXGI_FORMAT_R8G8B8A8_UNORM_SRGB`. Texture formats **should** be in order from highest to lowest runtime preference.

With an OpenGL-based graphics API, the texture formats correspond to OpenGL internal formats.

With a Direct3D-based graphics API, xrEnumerateSwapchainFormats never returns typeless formats (e.g. `DXGI_FORMAT_R8G8B8A8_TYPELESS`). Only concrete formats are returned, and only concrete formats may be specified by applications for swapchain creation.

Runtimes **must** always return identical buffer contents from this enumeration for the lifetime of the session.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- `formatCountOutput` **must** be a pointer to a `uint32_t` value
- If `formatCapacityInput` is not `0`, `formats` **must** be a pointer to an array of `formatCapacityInput` `int64_t` values

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_SIZE_INSUFFICIENT
- XR_ERROR_VALIDATION_FAILURE

XrSwapchainUsageFlags Specify the intended usage of the swapchain images. When images are created, the runtime needs to know how the images are used in a way that requires more information than simply the image format. The XrSwapchainCreateInfo passed to xrCreateSwapchain should match the intended usage or else undefined behavior may result when the application works with the images.

Flags include:

```
// Flag bits for XrSwapchainUsageFlags
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT = 0x00000001;
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT =
0x00000002;
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_UNORDERED_ACCESS_BIT = 0x00000004;
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_TRANSFER_SRC_BIT = 0x00000008;
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_TRANSFER_DST_BIT = 0x00000010;
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_SAMPLED_BIT = 0x00000020;
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT = 0x00000040;
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_MND =
0x00000080;
```

Flag Descriptions

- XR_SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT indicates that the image may be a color rendering target.
- XR_SWAPCHAIN_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT indicates that the image may be a depth/stencil rendering target.
- XR_SWAPCHAIN_USAGE_UNORDERED_ACCESS_BIT indicates that the image may be accessed out of order and that access may be via atomic operations.
- XR_SWAPCHAIN_USAGE_TRANSFER_SRC_BIT indicates that the image may be the source of a copy operation.
- XR_SWAPCHAIN_USAGE_TRANSFER_DST_BIT indicates that the image may be the destination of a copy operation.
- XR_SWAPCHAIN_USAGE_SAMPLED_BIT indicates that the image may be sampled by a shader.
- XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT indicates that the image format may be reinterpreted.

The xrCreateSwapchain function is defined as:

```
XrResult xrCreateSwapchain(
    XrSession                                   session,
    const XrSwapchainCreateInfo*                createInfo,
    XrSwapchain*                                swapchain);
```

Parameter Descriptions

- session is the session that creates the image.
- createInfo is a pointer to an XrSwapchainCreateInfo structure containing parameters to be used to create the image.
- swapchain is a pointer to a handle in which the created XrSwapchain is returned.

Creates an XrSwapchain handle. The returned swapchain handle **may** be subsequently used in API calls. Multiple XrSwapchain handles may exist simultaneously, up to some limit imposed by the runtime. The XrSwapchain handle **must** be eventually freed via the xrDestroySwapchain function. The runtime **must** return XR_ERROR_SWAPCHAIN_FORMAT_UNSUPPORTED if the image format specified in the XrSwapchainCreateInfo is unsupported. The runtime **must** return XR_ERROR_FEATURE_UNSUPPORTED if any bit of the create flags specified in the XrSwapchainCreateInfo is unsupported.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- `createInfo` **must** be a pointer to a valid XrSwapchainCreateInfo structure
- `swapchain` **must** be a pointer to an XrSwapchain handle

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_LIMIT_REACHED`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_OUT_OF_MEMORY`
- `XR_ERROR_SWAPCHAIN_FORMAT_UNSUPPORTED`
- `XR_ERROR_FEATURE_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`

The XrSwapchainCreateInfo structure is defined as:

```
typedef struct XrSwapchainCreateInfo {
    XrStructureType           type;
    const void*               next;
    XrSwapchainCreateFlags    createFlags;
    XrSwapchainUsageFlags     usageFlags;
    int64_t                   format;
    uint32_t                  sampleCount;
    uint32_t                  width;
    uint32_t                  height;
    uint32_t                  faceCount;
    uint32_t                  arraySize;
    uint32_t                  mipCount;
} XrSwapchainCreateInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `createFlags` is a bitmask of XrSwapchainCreateFlagBits describing additional properties of the swapchain.
- `usageFlags` is a bitmask of XrSwapchainUsageFlagBits describing the intended usage of the swapchain's images. The usage flags define how the corresponding graphics API objects are created. A mismatch **may** result in swapchain images that do not support the application's usage.
- `format` is a graphics API-specific texture format identifier. For example, if the graphics API specified in xrCreateSession is Vulkan, then this format is a Vulkan format such as `VK_FORMAT_R8G8B8A8_SRGB`. The format identifies the format that the runtime will interpret the texture as upon submission. Valid formats are indicated by xrEnumerateSwapchainFormats.
- `sampleCount` is the number of sub-data element samples in the image, **must** not be `0` or greater than the graphics API's maximum limit.
- `width` is the width of the image, **must** not be `0` or greater than the graphics API's maximum limit.
- `height` is the height of the image, **must** not be `0` or greater than the graphics API's maximum limit.
- `faceCount` is the number of faces, which can be either `6` (for cubemaps) or `1`.
- `arraySize` is the number of array layers in the image or `1` for a 2D image, **must** not be `0` or greater than the graphics API's maximum limit.
- `mipCount` describes the number of levels of detail available for minified sampling of the image, **must** not be `0` or greater than the graphics API's maximum limit.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_SWAPCHAIN_CREATE_INFO`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrAndroidSurfaceSwapchainCreateInfoFB, XrSecondaryViewConfigurationSwapchainCreateInfoMSFT
- `createFlags` **must** be `0` or a valid combination of XrSwapchainCreateFlagBits values
- `usageFlags` **must** be `0` or a valid combination of XrSwapchainUsageFlagBits values

The `createFlags` are a combination of the following:

```
// Flag bits for XrSwapchainCreateFlags
static const XrSwapchainCreateFlags XR_SWAPCHAIN_CREATE_PROTECTED_CONTENT_BIT =
0x00000001;
static const XrSwapchainCreateFlags XR_SWAPCHAIN_CREATE_STATIC_IMAGE_BIT = 0x00000002;
```

Flag Descriptions

- `XR_SWAPCHAIN_CREATE_PROTECTED_CONTENT_BIT` indicates that the swapchain's images will be protected from CPU access, using a mechanism such as Vulkan protected memory.
- `XR_SWAPCHAIN_CREATE_STATIC_IMAGE_BIT` indicates that the application will acquire and release only one image to this swapchain over its entire lifetime. The runtime **must** allocate only one swapchain image.

A runtime **may** implement any of these, but is not required to. A runtime **must** return `XR_ERROR_FEATURE_UNSUPPORTED` from xrCreateSwapchain if an XrSwapchainCreateFlags bit is requested but not implemented.

The number of images in each swapchain is implementation-defined except in the case of a static swapchain. To obtain the number of images actually allocated, call xrEnumerateSwapchainImages.

With a Direct3D-based graphics API, the swapchain returned by xrCreateSwapchain will be a typeless format if the requested format has a typeless analogue. Applications are required to reinterpret the swapchain as a compatible non-typeless type. Upon submitting such swapchains to the runtime, they are interpreted as the format specified by the application in the XrSwapchainCreateInfo.

Swapchains will be created with graphics API-specific flags appropriate to the type of underlying image and its usage. Extensions may exist to further assist the runtime in choosing how to create swapchains.

Runtimes **must** honor underlying graphics API limits when creating resources.

xrEnumerateSwapchainFormats never returns typeless formats (e.g. `DXGI_FORMAT_R8G8B8A8_TYPELESS`). Only concrete formats are returned, and only concrete formats may be specified by applications for swapchain creation.

The xrDestroySwapchain function is defined as:

```
XrResult xrDestroySwapchain(
    XrSwapchain                                 swapchain);
```

Parameter Descriptions

- `swapchain` is the swapchain to destroy.

All submitted graphics API commands that refer to `swapchain` **must** have completed execution. Runtimes **may** continue to utilize swapchain images after xrDestroySwapchain is called.

Valid Usage (Implicit)

- `swapchain` **must** be a valid XrSwapchain handle

Thread Safety

- Access to `swapchain`, and any child handles, **must** be externally synchronized

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_HANDLE_INVALID`

Swapchain images are acquired, waited on, and released by index, but the number of images in a swapchain is implementation-defined. Additionally, rendering to images requires access to the underlying image primitive of the graphics API being used. Applications may query and cache the images at any time after swapchain creation.

The xrEnumerateSwapchainImages function is defined as:

```
XrResult xrEnumerateSwapchainImages(
    XrSwapchain                                 swapchain,
    uint32_t                                    imageCapacityInput,
    uint32_t*                                   imageCountOutput,
    XrSwapchainImageBaseHeader*                 images);
```

Parameter Descriptions

- `swapchain` is the XrSwapchain to get images from.
- `imageCapacityInput` is the capacity of the `images` array, or 0 to indicate a request to retrieve the required capacity.
- `imageCountOutput` is a pointer to the count of `images` written, or a pointer to the required capacity in the case that `imageCapacityInput` is 0.
- `images` is a pointer to an array of graphics API-specific `XrSwapchainImage` structures, all of the same type, based on XrSwapchainImageBaseHeader. It **can** be `NULL` if `imageCapacityInput` is 0.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required `images` size.

Fills an array of graphics API-specific `XrSwapchainImage` structures. The resources **must** be constant and valid for the lifetime of the XrSwapchain.

Runtimes **must** always return identical buffer contents from this enumeration for the lifetime of the swapchain.

Note: `images` is a pointer to an array of structures of graphics API-specific type, not an array of structure pointers.

The pointer submitted as `images` will be treated as an array of the expected graphics API-specific type based on the graphics API used at session creation time. If the `type` member of any array element accessed in this way does not match the expected value, the runtime **must** return `XR_ERROR_VALIDATION_FAILURE`.

> *Note*
>
> Under a typical memory model, a runtime must treat the supplied pointer as an opaque blob beginning with XrSwapchainImageBaseHeader, until after it has verified the `type`.

Valid Usage (Implicit)

- `swapchain` **must** be a valid XrSwapchain handle
- `imageCountOutput` **must** be a pointer to a `uint32_t` value
- If `imageCapacityInput` is not `0`, `images` **must** be a pointer to an array of `imageCapacityInput` XrSwapchainImageBaseHeader-based structures. See also: XrSwapchainImageD3D11KHR, XrSwapchainImageD3D12KHR, XrSwapchainImageOpenGLESKHR, XrSwapchainImageOpenGLKHR, XrSwapchainImageVulkanKHR

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_SIZE_INSUFFICIENT
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_VALIDATION_FAILURE

The XrSwapchainImageBaseHeader structure is defined as:

```
typedef struct XrSwapchainImageBaseHeader {
    XrStructureType    type;
    void*              next;
} XrSwapchainImageBaseHeader;
```

Member Descriptions

- type is the XrStructureType of this structure. This base structure itself has no associated XrStructureType value.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

The XrSwapchainImageBaseHeader is a base structure that can be overridden by a graphics API-specific XrSwapchainImage* child structure.

Valid Usage (Implicit)

- `type` **must** be one of the following XrStructureType values: `XR_TYPE_SWAPCHAIN_IMAGE_D3D11_KHR`, `XR_TYPE_SWAPCHAIN_IMAGE_D3D12_KHR`, `XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_ES_KHR`, `XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_KHR`, `XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

Before an application can start building graphics API command buffers that refer to an image in a swapchain, it must acquire the image from the swapchain. The acquire operation determines the index of the next image that will be used in the swapchain. The order in which images are acquired is undefined. The runtime **must** allow the application to acquire more than one image from a single swapchain at a time, for example if the application implements a multiple frame deep rendering pipeline.

The xrAcquireSwapchainImage function is defined as:

```
XrResult xrAcquireSwapchainImage(
    XrSwapchain                                 swapchain,
    const XrSwapchainImageAcquireInfo*          acquireInfo,
    uint32_t*                                   index);
```

Parameter Descriptions

- `swapchain` is the swapchain from which to acquire an image.
- `acquireInfo` exists for extensibility purposes, it is `NULL` or a pointer to a valid XrSwapchainImageAcquireInfo.
- `index` is the returned image index that has been acquired.

Acquires the image corresponding to the `index` position in the array returned by xrEnumerateSwapchainImages. The runtime **must** return `XR_ERROR_CALL_ORDER_INVALID` if the next available index has already been acquired and not yet released with xrReleaseSwapchainImage. If the `swapchain` was created with the `XR_SWAPCHAIN_CREATE_STATIC_IMAGE_BIT` set in XrSwapchainCreateInfo ::`createFlags`, this function **must** not have been previously called for this swapchain. The runtime **must** return `XR_ERROR_CALL_ORDER_INVALID` if a `swapchain` created with the `XR_SWAPCHAIN_CREATE_STATIC_IMAGE_BIT` set in XrSwapchainCreateInfo::`createFlags` and this function has been successfully called previously for this swapchain.

Valid Usage (Implicit)

- `swapchain` **must** be a valid XrSwapchain handle
- If `acquireInfo` is not `NULL`, `acquireInfo` **must** be a pointer to a valid XrSwapchainImageAcquireInfo structure
- `index` **must** be a pointer to a `uint32_t` value

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_CALL_ORDER_INVALID`

The XrSwapchainImageAcquireInfo structure is defined as:

```
typedef struct XrSwapchainImageAcquireInfo {
    XrStructureType    type;
    const void*        next;
} XrSwapchainImageAcquireInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

Because this structure only exists to support extension-specific structures, xrAcquireSwapchainImage will accept a `NULL` argument for `acquireInfo` for applications that are not using any relevant extensions.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_SWAPCHAIN_IMAGE_ACQUIRE_INFO`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

The xrWaitSwapchainImage function is defined as:

```
XrResult xrWaitSwapchainImage(
    XrSwapchain                                 swapchain,
    const XrSwapchainImageWaitInfo*             waitInfo);
```

Parameter Descriptions

- `swapchain` is the swapchain from which to wait for an image.
- `waitInfo` is a pointer to an XrSwapchainImageWaitInfo structure.

Before an application can begin writing to a swapchain image, it must first wait on the image to avoid writing to it before the compositor has finished reading from it. xrWaitSwapchainImage will implicitly wait on the oldest acquired swapchain image which has not yet been successfully waited on. Once a swapchain image has been successfully waited on, it **must** be released before waiting on the next acquired swapchain image.

This function may block for longer than the timeout specified in XrSwapchainImageWaitInfo due to scheduling or contention.

If the timeout expires without the image becoming available for writing, `XR_TIMEOUT_EXPIRED` **must** be returned. If xrWaitSwapchainImage returns `XR_TIMEOUT_EXPIRED`, the next call to xrWaitSwapchainImage will wait on the same image index again until the function succeeds with `XR_SUCCESS`. Note that this is not an error code; `XR_SUCCEEDED(XR_TIMEOUT_EXPIRED)` is `true`.

The runtime **must** eventually relinquish ownership of a swapchain image to the application and **must** not block indefinitely.

The runtime **must** return `XR_ERROR_CALL_ORDER_INVALID` if no image has been acquired by calling xrAcquireSwapchainImage.

Valid Usage (Implicit)

- `swapchain` **must** be a valid XrSwapchain handle
- `waitInfo` **must** be a pointer to a valid XrSwapchainImageWaitInfo structure

Return Codes

Success

- `XR_SUCCESS`
- `XR_TIMEOUT_EXPIRED`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_CALL_ORDER_INVALID`

The XrSwapchainImageWaitInfo structure describes a swapchain image wait operation. It is defined as:

```
typedef struct XrSwapchainImageWaitInfo {
    XrStructureType    type;
    const void*        next;
    XrDuration         timeout;
} XrSwapchainImageWaitInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `timeout` indicates how many nanoseconds the call should block waiting for the image to become available for writing.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_SWAPCHAIN_IMAGE_WAIT_INFO`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

Once an application is done submitting commands that reference the swapchain image, the application **must** release the swapchain image. xrReleaseSwapchainImage will implicitly release the oldest swapchain image which has been acquired. The swapchain image **must** have been successfully waited on before it is released. xrEndFrame will use the most recently released swapchain image. In each frame submitted to the compositor only one image index from each swapchain will be used. Note that in case the swapchain contains 2D image arrays, one array is referenced per swapchain index and thus the whole image array can be used in one frame.

The xrReleaseSwapchainImage function is defined as:

```
XrResult xrReleaseSwapchainImage(
    XrSwapchain                                 swapchain,
    const XrSwapchainImageReleaseInfo*          releaseInfo);
```

Parameter Descriptions

- `swapchain` is the XrSwapchain from which to release an image.
- `releaseInfo` exists for extensibility purposes, it is `NULL` or a pointer to a valid XrSwapchainImageReleaseInfo.

If the `swapchain` was created with the `XR_SWAPCHAIN_CREATE_STATIC_IMAGE_BIT` set in XrSwapchainCreateInfo::`createFlags` structure, this function **must** not have been previously called for this swapchain.

The runtime **must** return `XR_ERROR_CALL_ORDER_INVALID` if no image has been waited on by calling xrWaitSwapchainImage.

Valid Usage (Implicit)

- `swapchain` **must** be a valid XrSwapchain handle
- If `releaseInfo` is not `NULL`, `releaseInfo` **must** be a pointer to a valid XrSwapchainImageReleaseInfo structure

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_CALL_ORDER_INVALID`

The XrSwapchainImageReleaseInfo structure is defined as:

```
typedef struct XrSwapchainImageReleaseInfo {
    XrStructureType    type;
    const void*        next;
} XrSwapchainImageReleaseInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

Because this structure only exists to support extension-specific structures, xrReleaseSwapchainImage will accept a NULL argument for releaseInfo for applications that are not using any relevant extensions.

Valid Usage (Implicit)

- type **must** be XR_TYPE_SWAPCHAIN_IMAGE_RELEASE_INFO
- next **must** be NULL or a valid pointer to the next structure in a structure chain

10.2. View and Projection State

An application uses xrLocateViews to retrieve the viewer pose and projection parameters needed to render each view for use in a composition projection layer.

The xrLocateViews function is defined as:

```
XrResult xrLocateViews(
    XrSession                                   session,
    const XrViewLocateInfo*                     viewLocateInfo,
    XrViewState*                                viewState,
    uint32_t                                    viewCapacityInput,
    uint32_t*                                   viewCountOutput,
    XrView*                                     views);
```

Parameter Descriptions

- session is a handle to the provided XrSession.
- viewLocateInfo is a pointer to a valid XrViewLocateInfo structure.
- viewState is the output structure with the viewer state information.
- viewCapacityInput is an input parameter which specifies the capacity of the views array. The required capacity **must** be same as defined by the corresponding XrViewConfigurationType.
- viewCountOutput is an output parameter which identifies the valid count of views.
- views is an array of XrView.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required views size.

The xrLocateViews function returns the view and projection info for a particular display time. This time is typically the target display time for a given frame. Repeatedly calling xrLocateViews with the same time **may** not necessarily return the same result. Instead the prediction gets increasingly accurate as the function is called closer to the given time for which a prediction is made. This allows an application to get the predicted views as late as possible in its pipeline to get the least amount of latency and prediction error.

xrLocateViews returns an array of XrView elements, one for each view of the specified view configuration type, along with an XrViewState containing additional state data shared across all views. The eye each view corresponds to is statically defined in XrViewConfigurationType in case the application wants to apply eye-specific rendering traits. The XrViewState and XrView member data may change on subsequent calls to xrLocateViews, and so applications **must** not assume it to be constant.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- `viewLocateInfo` **must** be a pointer to a valid XrViewLocateInfo structure
- `viewState` **must** be a pointer to an XrViewState structure
- `viewCountOutput` **must** be a pointer to a `uint32_t` value
- If `viewCapacityInput` is not `0`, `views` **must** be a pointer to an array of `viewCapacityInput` XrView structures

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_SIZE_INSUFFICIENT`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED`
- `XR_ERROR_TIME_INVALID`

The XrViewLocateInfo structure is defined as:

```
typedef struct XrViewLocateInfo {
    XrStructureType           type;
    const void*               next;
    XrViewConfigurationType   viewConfigurationType;
    XrTime                    displayTime;
    XrSpace                   space;
} XrViewLocateInfo;
```

Member Descriptions

- `viewConfigurationType` is XrViewConfigurationType to query for.
- `displayTime` is the time for which the view poses are predicted.
- `space` is the XrSpace in which the `pose` in each XrView is expressed.

The XrViewLocateInfo structure contains the display time and space used to locate the view XrView structures.

The runtime **must** return error `XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED` if the given `viewConfigurationType` is not one of the supported type reported by xrEnumerateViewConfigurations.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_VIEW_LOCATE_INFO`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrViewLocateFoveatedRenderingVARJO
- `viewConfigurationType` **must** be a valid XrViewConfigurationType value
- `space` **must** be a valid XrSpace handle

The XrView structure is defined as:

```
typedef struct XrView {
    XrStructureType    type;
    void*              next;
    XrPosef            pose;
    XrFovf             fov;
} XrView;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `pose` is an XrPosef defining the location and orientation of the view in the `space` specified by the xrLocateViews function.
- `fov` is the XrFovf for the four sides of the projection.

The XrView structure contains view pose and projection state necessary to render a single projection view in the view configuration.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_VIEW`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

The XrViewState structure is defined as:

```
typedef struct XrViewState {
    XrStructureType     type;
    void*               next;
    XrViewStateFlags    viewStateFlags;
} XrViewState;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `viewStateFlags` is a bitmask of XrViewStateFlagBits indicating state for all views.

The XrViewState contains additional view state from xrLocateViews common to all views of the active view configuration.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_VIEW_STATE`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `viewStateFlags` **must** be `0` or a valid combination of XrViewStateFlagBits values

The XrViewStateFlags specifies the validity and quality of the corresponding XrView array returned by xrLocateViews.

Flags include:

```
// Flag bits for XrViewStateFlags
static const XrViewStateFlags XR_VIEW_STATE_ORIENTATION_VALID_BIT = 0x00000001;
static const XrViewStateFlags XR_VIEW_STATE_POSITION_VALID_BIT = 0x00000002;
static const XrViewStateFlags XR_VIEW_STATE_ORIENTATION_TRACKED_BIT = 0x00000004;
static const XrViewStateFlags XR_VIEW_STATE_POSITION_TRACKED_BIT = 0x00000008;
```

Flag Descriptions

- `XR_VIEW_STATE_ORIENTATION_VALID_BIT` indicates whether all XrView orientations contain valid data. Applications **must** not read any of the XrView `pose` `orientation` fields if this flag is unset. `XR_VIEW_STATE_ORIENTATION_TRACKED_BIT` **should** generally remain set when this bit is set for views on a tracked headset or handheld device.
- `XR_VIEW_STATE_POSITION_VALID_BIT` indicates whether all XrView positions contain valid data. When a view loses tracking, runtimes **should** continue to provide valid but untracked view `position` values that are inferred or last-known, so long as it's still meaningful for the application to render content using that position, clearing `XR_VIEW_STATE_POSITION_TRACKED_BIT` until tracking is recovered. Applications **must** not read any of the XrView `pose` `position` fields if this flag is unset.
- `XR_VIEW_STATE_ORIENTATION_TRACKED_BIT` indicates whether all XrView orientations represent an actively tracked orientation. This bit **should** generally remain set when `XR_VIEW_STATE_ORIENTATION_VALID_BIT` is set for views on a tracked headset or handheld device.
- `XR_VIEW_STATE_POSITION_TRACKED_BIT` indicates whether all XrView positions represent an actively tracked position. When a view loses tracking, runtimes **should** continue to provide valid but untracked view `position` values that are inferred or last-known, e.g. based on neck model updates, inertial dead reckoning, or a last-known position, so long as it's still meaningful for the application to render content using that position.

10.3. Frame Synchronization

An application synchronizes its rendering loop to the runtime by calling xrWaitFrame.

The xrWaitFrame function is defined as:

```
XrResult xrWaitFrame(
    XrSession                                   session,
    const XrFrameWaitInfo*                      frameWaitInfo,
    XrFrameState*                               frameState);
```

Parameter Descriptions

- `session` is a valid XrSession handle.
- `frameWaitInfo` exists for extensibility purposes, it is `NULL` or a pointer to a valid XrFrameWaitInfo.
- `frameState` is a pointer to a valid XrFrameState, an output parameter.

xrWaitFrame throttles the application frame loop in order to synchronize application frame submissions with the display. xrWaitFrame returns a predicted display time for the next time that the runtime predicts a composited frame will be displayed. The runtime **may** affect this computation by changing the return values and throttling of xrWaitFrame in response to feedback from frame submission and completion times in xrEndFrame. An application **must** eventually match each xrWaitFrame call with one call to xrBeginFrame. A subsequent xrWaitFrame call **must** block until the previous frame has been begun with xrBeginFrame and **must** unblock independently of the corresponding call to xrEndFrame. When less than one frame interval has passed since the previous return from xrWaitFrame, the runtime **should** block until the beginning of the next frame interval. If more than one frame interval has passed since the last return from xrWaitFrame, the runtime **may** return immediately or block until the beginning of the next frame interval.

In the case that an application has pipelined frame submissions, the application **should** compute the appropriate target display time using both the predicted display time and predicted display interval. The application **should** use the computed target display time when requesting space and view locations for rendering.

The XrFrameState::`predictedDisplayTime` returned by xrWaitFrame **must** be monotonically increasing.

The runtime **may** dynamically adjust the start time of the frame interval relative to the display hardware's refresh cycle to minimize graphics processor contention between the application and the compositor.

xrWaitFrame **must** be callable from any thread, including a different thread than xrBeginFrame /xrEndFrame are being called from.

Calling xrWaitFrame **must** be externally synchronized by the application, concurrent calls **may** result in undefined behavior.

The runtime **must** return `XR_ERROR_SESSION_NOT_RUNNING` if the `session` is not running.

> *Note*
>
> The engine simulation **should** advance based on the display time. Every stage in the engine pipeline should use the exact same display time for one particular application-generated frame. An accurate and consistent display time across all stages and threads in the engine pipeline is important to avoid object motion judder. If the application has multiple pipeline stages, the application should pass its computed display time through its pipeline, as xrWaitFrame must be called only once per frame.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- If `frameWaitInfo` is not `NULL`, `frameWaitInfo` **must** be a pointer to a valid XrFrameWaitInfo structure
- `frameState` **must** be a pointer to an XrFrameState structure

Thread Safety

- Access to the `session` parameter by any other xrWaitFrame call **must** be externally synchronized

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_SESSION_NOT_RUNNING
- XR_ERROR_VALIDATION_FAILURE

The XrFrameWaitInfo structure is defined as:

```
typedef struct XrFrameWaitInfo {
    XrStructureType    type;
    const void*        next;
} XrFrameWaitInfo;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

Because this structure only exists to support extension-specific structures, xrWaitFrame **must** accept a NULL argument for frameWaitInfo for applications that are not using any relevant extensions.

Valid Usage (Implicit)

- type **must** be XR_TYPE_FRAME_WAIT_INFO
- next **must** be NULL or a valid pointer to the next structure in a structure chain

The XrFrameState structure is defined as:

```
typedef struct XrFrameState {
    XrStructureType    type;
    void*              next;
    XrTime             predictedDisplayTime;
    XrDuration         predictedDisplayPeriod;
    XrBool32           shouldRender;
} XrFrameState;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `predictedDisplayTime` is the anticipated display `XrTime` for the next application-generated frame.
- `predictedDisplayPeriod` is the `XrDuration` of the display period for the next application-generated frame, for use in predicting display times beyond the next one.
- `shouldRender` is `XR_TRUE` if the application **should** render its layers as normal and submit them to xrEndFrame. When this value is `XR_FALSE`, the application **should** avoid heavy GPU work where possible, for example by skipping layer rendering and then omitting those layers when calling xrEndFrame.

XrFrameState describes the time at which the next frame will be displayed to the user. `predictedDisplayTime` **must** refer to the midpoint of the interval during which the frame is displayed. The runtime **may** report a different `predictedDisplayPeriod` from the hardware's refresh cycle.

For any frame where `shouldRender` is `XR_FALSE`, the application **should** avoid heavy GPU work for that frame, for example by not rendering its layers. This typically happens when the application is transitioning into or out of a running session, or when some system UI is fully covering the application at the moment. As long as the session is running, the application **should** keep running the frame loop to maintain the frame synchronization to the runtime, even if this requires calling xrEndFrame with all layers omitted.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_FRAME_STATE`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrSecondaryViewConfigurationFrameStateMSFT

10.4. Frame Submission

Every application **must** call xrBeginFrame before calling xrEndFrame, and **should** call xrEndFrame before calling xrBeginFrame again. Calling xrEndFrame again without a prior call to xrBeginFrame **must** result in XR_ERROR_CALL_ORDER_INVALID being returned by xrEndFrame. An application **may** call xrBeginFrame again if the prior xrEndFrame fails or if the application wishes to discard an in-progress frame. A successful call to xrBeginFrame again with no intervening xrEndFrame call **must** result in the success code XR_FRAME_DISCARDED being returned from xrBeginFrame. In this case it is assumed that the xrBeginFrame refers to the next frame and the previously begun frame is forfeited by the application. An application **may** call xrEndFrame without having called xrReleaseSwapchainImage since the previous call to xrEndFrame for any swapchain passed to xrEndFrame. Applications **should** call xrBeginFrame right before executing any graphics device work for a given frame, as opposed to calling it afterwards. The runtime **must** only compose frames whose xrBeginFrame and xrEndFrame both return success codes. While xrBeginFrame and xrEndFrame do not need to be called on the same thread, the application **must** handle synchronization if they are called on separate threads.

The xrBeginFrame function is defined as:

```
XrResult xrBeginFrame(
    XrSession                                   session,
    const XrFrameBeginInfo*                     frameBeginInfo);
```

Parameter Descriptions

- session is a valid XrSession handle.
- frameBeginInfo exists for extensibility purposes, it is NULL or a pointer to a valid XrFrameBeginInfo.

xrBeginFrame is called prior to the start of frame rendering. The application **should** still call xrBeginFrame but omit rendering work for the frame if XrFrameState::shouldRender is XR_FALSE.

The runtime **must** return the error code XR_ERROR_CALL_ORDER_INVALID if there was no corresponding successful call to xrWaitFrame.

The runtime **must** return the success code XR_FRAME_DISCARDED if a prior xrBeginFrame has been called without an intervening call to xrEndFrame.

The runtime **must** return XR_ERROR_SESSION_NOT_RUNNING if the session is not running.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- If `frameBeginInfo` is not `NULL`, `frameBeginInfo` **must** be a pointer to a valid XrFrameBeginInfo structure

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`
- `XR_FRAME_DISCARDED`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_CALL_ORDER_INVALID`
- `XR_ERROR_SESSION_NOT_RUNNING`
- `XR_ERROR_VALIDATION_FAILURE`

The XrFrameBeginInfo structure is defined as:

```
typedef struct XrFrameBeginInfo {
    XrStructureType    type;
    const void*        next;
} XrFrameBeginInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

Because this structure only exists to support extension-specific structures, xrBeginFrame will accept a NULL argument for frameBeginInfo for applications that are not using any relevant extensions.

Valid Usage (Implicit)

- type **must** be XR_TYPE_FRAME_BEGIN_INFO
- next **must** be NULL or a valid pointer to the next structure in a structure chain

The xrEndFrame function is defined as:

```
XrResult xrEndFrame(
    XrSession                                   session,
    const XrFrameEndInfo*                       frameEndInfo);
```

Parameter Descriptions

- session is a valid XrSession handle.
- frameEndInfo is a pointer to a valid XrFrameEndInfo.

xrEndFrame **may** return immediately to the application. XrFrameEndInfo::displayTime **should** be computed using values returned by xrWaitFrame. The runtime **should** be robust against variations in the timing of calls to xrWaitFrame, since a pipelined system may call xrWaitFrame on a separate thread from xrBeginFrame and xrEndFrame without any synchronization guarantees.

> *Note*
>
> An accurate predicted display time is very important to avoid black pull-in by reprojection and to reduce motion judder in case the runtime does not implement a translational reprojection. Reprojection should never display images before the display refresh period they were predicted for, even if they are completed early, because this will cause motion judder just the same. In other words, the better the predicted display time, the less latency experienced by the user.

Every call to xrEndFrame **must** be preceded by a successful call to xrBeginFrame. Failure to do so **must** result in XR_ERROR_CALL_ORDER_INVALID being returned by xrEndFrame. XrFrameEndInfo **may** reference swapchains into which the application has rendered for this frame. From each XrSwapchain only one image index is implicitly referenced per frame, the one corresponding to the last call to xrReleaseSwapchainImage. However, a specific swapchain (and by extension a specific swapchain image index) **may** be referenced in XrFrameEndInfo multiple times. This can be used for example to render a side by side image into a single swapchain image and referencing it twice with differing image rectangles in different layers.

If no layers are provided then the display **must** be cleared.

XR_ERROR_LAYER_INVALID **must** be returned if an unknown, unsupported layer type, or NULL pointer is passed as one of the XrFrameEndInfo::layers.

XR_ERROR_LAYER_INVALID **must** be returned if a layer references a swapchain that has no released swapchain image.

XR_ERROR_LAYER_LIMIT_EXCEEDED **must** be returned if XrFrameEndInfo::layerCount exceeds XrSystemGraphicsProperties::maxLayerCount or if the runtime is unable to composite the specified layers due to resource constraints.

XR_ERROR_SWAPCHAIN_RECT_INVALID **must** be returned if XrFrameEndInfo::layers contains a composition layer which references pixels outside of the associated swapchain image or if negatively sized.

XR_ERROR_ENVIRONMENT_BLEND_MODE_UNSUPPORTED **must** be returned if XrFrameEndInfo::environmentBlendMode is not supported.

XR_ERROR_SESSION_NOT_RUNNING **must** be returned if the session is not running.

> *Note*
>
> Applications should discard frames for which xrEndFrame returns a recoverable error over attempting to resubmit the frame with different frame parameters to provide a more consistent experience across different runtime implementations.

Valid Usage (Implicit)

- session **must** be a valid XrSession handle
- frameEndInfo **must** be a pointer to a valid XrFrameEndInfo structure

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_CALL_ORDER_INVALID
- XR_ERROR_LAYER_INVALID
- XR_ERROR_SWAPCHAIN_RECT_INVALID
- XR_ERROR_ENVIRONMENT_BLEND_MODE_UNSUPPORTED
- XR_ERROR_SESSION_NOT_RUNNING
- XR_ERROR_LAYER_LIMIT_EXCEEDED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_TIME_INVALID
- XR_ERROR_POSE_INVALID

The XrFrameEndInfo structure is defined as:

```
typedef struct XrFrameEndInfo {
    XrStructureType                               type;
    const void*                                   next;
    XrTime                                        displayTime;
    XrEnvironmentBlendMode                        environmentBlendMode;
    uint32_t                                      layerCount;
    const XrCompositionLayerBaseHeader* const*    layers;
} XrFrameEndInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `displayTime` is the `XrTime` at which this frame **should** be displayed.
- `environmentBlendMode` is the XrEnvironmentBlendMode value representing the desired environment blend mode for this frame.
- `layerCount` is the number of composition layers in this frame. The maximum supported layer count is identified by XrSystemGraphicsProperties::maxLayerCount. If layerCount is greater than the maximum supported layer count then `XR_ERROR_LAYER_LIMIT_EXCEEDED` **must** be returned.
- `layers` is a pointer to an array of XrCompositionLayerBaseHeader pointers.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_FRAME_END_INFO`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrSecondaryViewConfigurationFrameEndInfoMSFT
- `environmentBlendMode` **must** be a valid XrEnvironmentBlendMode value
- If `layerCount` is not `0`, `layers` **must** be a pointer to an array of `layerCount` valid XrCompositionLayerBaseHeader-based structures. See also: XrCompositionLayerCubeKHR, XrCompositionLayerCylinderKHR, XrCompositionLayerEquirect2KHR, XrCompositionLayerEquirectKHR, XrCompositionLayerProjection, XrCompositionLayerQuad

All layers submitted to xrEndFrame will be presented to the primary view configuration of the running session.

10.4.1. Frame Rate

For every application-generated frame, the application **may** call xrEndFrame to submit the application-generated composition layers. In addition, the application **must** call xrWaitFrame when the application is ready to begin preparing the next set of frame layers. xrEndFrame **may** return immediately to the application, but xrWaitFrame **must** block for an amount of time that depends on throttling of the application by the runtime. The earliest the runtime will return from xrWaitFrame is when it determines that the application **should** start drawing the next frame.

10.4.2. Compositing

Composition layers are submitted by the application via the xrEndFrame call. All composition layers to be drawn **must** be submitted with every xrEndFrame call. A layer that is omitted in this call will not be drawn by the runtime layer compositor. All views associated with projection layers **must** be supplied, or `XR_ERROR_VALIDATION_FAILURE` **must** be returned by xrEndFrame.

Composition layers **must** be drawn in the same order as they are specified in via XrFrameEndInfo, with the 0th layer drawn first. Layers **must** be drawn with a "painter's algorithm," with each successive layer potentially overwriting the destination layers whether or not the new layers are virtually closer to the viewer.

10.4.3. Composition Layer Flags

The XrCompositionLayerFlagBits bitfield is specified as:

```
// Flag bits for XrCompositionLayerFlags
static const XrCompositionLayerFlags
XR_COMPOSITION_LAYER_CORRECT_CHROMATIC_ABERRATION_BIT = 0x00000001;
static const XrCompositionLayerFlags XR_COMPOSITION_LAYER_BLEND_TEXTURE_SOURCE_ALPHA_BIT
= 0x00000002;
static const XrCompositionLayerFlags XR_COMPOSITION_LAYER_UNPREMULTIPLIED_ALPHA_BIT =
0x00000004;
```

XrCompositionLayerFlags specify options for individual composition layers.

Flag Descriptions

- `XR_COMPOSITION_LAYER_CORRECT_CHROMATIC_ABERRATION_BIT` enables optical chromatic aberration correction for the layer when not done by default.
- `XR_COMPOSITION_LAYER_BLEND_TEXTURE_SOURCE_ALPHA_BIT` enables the layer's texture alpha channel.
- `XR_COMPOSITION_LAYER_UNPREMULTIPLIED_ALPHA_BIT` indicates that the layer's color components have not been premultiplied with the layer's alpha component.

10.4.4. Composition Layer Blending

All types of composition layers are subject to blending with other layers. Blending of layers can be controlled by layer per-texel source alpha. Layer swapchain textures may contain an alpha channel, depending on the image format. If a submitted swapchain's texture format does not include an alpha channel or if the `XR_COMPOSITION_LAYER_BLEND_TEXTURE_SOURCE_ALPHA_BIT` is unset, then the layer alpha is initialized to one.

If the swapchain texture format color encoding is other than RGBA, it is converted to RGBA.

If the texture color channels are encoded without premultiplying by alpha, the XR_COMPOSITION_LAYER_UNPREMULTIPLIED_ALPHA_BIT **should** be set. The effect of this bit alters the layer color as follows:

```
LayerColor.RGB *= LayerColor.A
```

LayerColor is then clamped to a range of [0.0, 1.0].

The layer blending operation is defined as:

```
CompositeColor  = LayerColor + CompositeColor * (1 - LayerColor.A)
```

Before the first layer is composited, all components of CompositeColor are initialized to zero.

10.4.5. Composition Layer Types

Composition layers allow an application to offload the composition of the final image to a runtime-supplied compositor. This reduces the application's rendering complexity since details such as frame-rate interpolation and distortion correction can be performed by the runtime. The core specification defines XrCompositionLayerProjection and XrCompositionLayerQuad layer types.

The projection layer type represents planar projected images rendered from the eye point of each eye using a perspective projection. This layer type is typically used to render the virtual world from the user's perspective.

The quad layer type describes a posable planar rectangle in the virtual world for displaying two-dimensional content. Quad layers can subtend a smaller portion of the display's field of view, allowing a better match between the resolutions of the XrSwapchain image and footprint of that image in the final composition. This improves legibility for user interface elements or heads-up displays and allows optimal sampling during any composition distortion corrections the runtime might employ.

The classes below describe the layer types in the layer composition system.

The XrCompositionLayerBaseHeader structure is defined as:

```
typedef struct XrCompositionLayerBaseHeader {
    XrStructureType            type;
    const void*                next;
    XrCompositionLayerFlags    layerFlags;
    XrSpace                    space;
} XrCompositionLayerBaseHeader;
```

Member Descriptions

- type is the XrStructureType of this structure. This base structure itself has no associated XrStructureType value.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- layerFlags is a bitmask of XrCompositionLayerFlagBits describing flags to apply to the layer.
- space is the XrSpace in which the layer will be kept stable over time.

All composition layer structures begin with the elements described in the XrCompositionLayerBaseHeader. The XrCompositionLayerBaseHeader structure is not intended to be directly used, but forms a basis for defining current and future structures containing composition layer information. The XrFrameEndInfo structure contains an array of pointers to these polymorphic header structures. All composition layer type pointers **must** be type-castable as an XrCompositionLayerBaseHeader pointer.

Valid Usage (Implicit)

- type **must** be one of the following XrStructureType values: XR_TYPE_COMPOSITION_LAYER_CUBE_KHR, XR_TYPE_COMPOSITION_LAYER_CYLINDER_KHR, XR_TYPE_COMPOSITION_LAYER_EQUIRECT2_KHR, XR_TYPE_COMPOSITION_LAYER_EQUIRECT_KHR, XR_TYPE_COMPOSITION_LAYER_PROJECTION, XR_TYPE_COMPOSITION_LAYER_QUAD
- next **must** be NULL or a valid pointer to the next structure in a structure chain. See also: XrCompositionLayerColorScaleBiasKHR
- layerFlags **must** be 0 or a valid combination of XrCompositionLayerFlagBits values
- space **must** be a valid XrSpace handle

Many composition layer structures also contain one or more references to generic layer data stored in an XrSwapchainSubImage structure.

The XrSwapchainSubImage structure is defined as:

```
typedef struct XrSwapchainSubImage {
    XrSwapchain    swapchain;
    XrRect2Di      imageRect;
    uint32_t       imageArrayIndex;
} XrSwapchainSubImage;
```

Member Descriptions

- `swapchain` is the XrSwapchain to be displayed.
- `imageRect` is an XrRect2Di representing the valid portion of the image to use, in pixels. It also implicitly defines the transform from normalized image coordinates into pixel coordinates. Note that the compositor **may** bleed in pixels from outside the bounds in some cases, for instance due to mipmapping.
- `imageArrayIndex` is the image array index, with 0 meaning the first or only array element.

Valid Usage (Implicit)

- `swapchain` **must** be a valid XrSwapchain handle

Projection Composition

The XrCompositionLayerProjection layer represents planar projected images rendered from the eye point of each eye using a standard perspective projection.

The XrCompositionLayerProjection structure is defined as:

```
typedef struct XrCompositionLayerProjection {
    XrStructureType                           type;
    const void*                               next;
    XrCompositionLayerFlags                   layerFlags;
    XrSpace                                   space;
    uint32_t                                  viewCount;
    const XrCompositionLayerProjectionView*    views;
} XrCompositionLayerProjection;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `layerFlags` is a bitmask of XrCompositionLayerFlagBits describing flags to apply to the layer.
- `space` is the XrSpace in which the `pose` of each XrCompositionLayerProjectionView is evaluated over time by the compositor.
- `viewCount` is the count of views in the `views` array. This **must** be equal to the number of view poses returned by xrLocateViews.
- `views` is the array of type XrCompositionLayerProjectionView containing each projection layer view.

Note

Because a runtime may reproject the layer over time, a projection layer should specify an XrSpace in which to maximize stability of the layer content. For example, a projection layer containing world-locked content should use an XrSpace which is also world-locked, such as the `LOCAL` or `STAGE` reference spaces. In the case that the projection layer should be head-locked, such as a heads up display, the `VIEW` reference space would provide the highest quality layer reprojection.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_COMPOSITION_LAYER_PROJECTION`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrCompositionLayerDepthTestVARJO
- `layerFlags` **must** be `0` or a valid combination of XrCompositionLayerFlagBits values
- `space` **must** be a valid XrSpace handle
- `views` **must** be a pointer to an array of `viewCount` valid XrCompositionLayerProjectionView structures
- The `viewCount` parameter **must** be greater than `0`

The XrCompositionLayerProjectionView structure is defined as:

```
typedef struct XrCompositionLayerProjectionView {
    XrStructureType        type;
    const void*            next;
    XrPosef                pose;
    XrFovf                 fov;
    XrSwapchainSubImage    subImage;
} XrCompositionLayerProjectionView;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- pose is an XrPosef defining the location and orientation of this projection element in the space of the corresponding XrCompositionLayerProjectionView.
- fov is the XrFovf for this projection element.
- subImage is the image layer XrSwapchainSubImage to use.

The count and order of view poses submitted with XrCompositionLayerProjection **must** be the same order as that returned by xrLocateViews. The XrCompositionLayerProjectionView::pose and XrCompositionLayerProjectionView::fov **should** almost always derive from XrView::pose and XrView ::fov as found in the xrLocateViews::views array. However, applications **may** submit an XrCompositionLayerProjectionView which has a different view or FOV than that from xrLocateViews. In this case, the runtime will map the view and FOV to the system display appropriately. In the case that two submitted views within a single layer overlap, they **must** be composited in view array order.

Valid Usage (Implicit)

- type **must** be XR_TYPE_COMPOSITION_LAYER_PROJECTION_VIEW
- next **must** be NULL or a valid pointer to the next structure in a structure chain. See also: XrCompositionLayerDepthInfoKHR
- subImage **must** be a valid XrSwapchainSubImage structure

Quad Layer Composition

The XrCompositionLayerQuad structure defined as:

```
typedef struct XrCompositionLayerQuad {
    XrStructureType            type;
    const void*                next;
    XrCompositionLayerFlags    layerFlags;
    XrSpace                    space;
    XrEyeVisibility            eyeVisibility;
    XrSwapchainSubImage        subImage;
    XrPosef                    pose;
    XrExtent2Df                size;
} XrCompositionLayerQuad;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `layerFlags` is a bitmask of XrCompositionLayerFlagBits describing flags to apply to the layer.
- `space` is the XrSpace in which the `pose` of the quad layer is evaluated over time.
- `eyeVisibility` is the XrEyeVisibility for this layer.
- `subImage` is the image layer XrSwapchainSubImage to use.
- `pose` is an XrPosef defining the position and orientation of the quad in the reference frame of the `space`.
- `size` is the width and height of the quad in meters.

The XrCompositionLayerQuad layer is useful for user interface elements or 2D content rendered into the virtual world. The layer's XrSwapchainSubImage::swapchain image is applied to a quad in the virtual world space. Only front face of the quad surface is visible; the back face is not visible and **must** not be drawn by the runtime. A quad layer has no thickness; it is a two-dimensional object positioned and oriented in 3D space. The position of a quad refers to the center of the quad within the given XrSpace. The orientation of the quad refers to the orientation of the normal vector from the front face. The size of a quad refers to the quad's size in the x-y plane of the given XrSpace's coordinate system. A quad with a position of {0,0,0}, rotation of {0,0,0,1} (no rotation), and a size of {1,1} refers to a 1 meter x 1 meter quad centered at {0,0,0} with its front face normal vector coinciding with the +z axis.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_COMPOSITION_LAYER_QUAD`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `layerFlags` **must** be `0` or a valid combination of XrCompositionLayerFlagBits values
- `space` **must** be a valid XrSpace handle
- `eyeVisibility` **must** be a valid XrEyeVisibility value
- `subImage` **must** be a valid XrSwapchainSubImage structure

The XrEyeVisibility enum selects which of the viewer's eyes to display a layer to:

```
typedef enum XrEyeVisibility {
    XR_EYE_VISIBILITY_BOTH = 0,
    XR_EYE_VISIBILITY_LEFT = 1,
    XR_EYE_VISIBILITY_RIGHT = 2,
    XR_EYE_VISIBILITY_MAX_ENUM = 0x7FFFFFFF
} XrEyeVisibility;
```

Enumerant Descriptions

- `XR_EYE_VISIBILITY_BOTH` displays the layer to both eyes.
- `XR_EYE_VISIBILITY_LEFT` displays the layer to the viewer's physical left eye.
- `XR_EYE_VISIBILITY_RIGHT` displays the layer to the viewer's physical right eye.

10.4.6. Environment Blend Mode

After the compositor has blended and flattened all layers (including any layers added by the runtime itself), it will then present this image to the system's display. The composited image will then blend with the user's view of the physical world behind the displays in one of three modes, based on the application's chosen **environment blend mode**. VR applications will generally choose the `XR_ENVIRONMENT_BLEND_MODE_OPAQUE` blend mode, while AR applications will generally choose either the `XR_ENVIRONMENT_BLEND_MODE_ADDITIVE` or `XR_ENVIRONMENT_BLEND_MODE_ALPHA_BLEND` mode.

Applications select their environment blend mode each frame as part of their call to xrEndFrame. The application can inspect the set of supported environment blend modes for a given system using xrEnumerateEnvironmentBlendModes, and prepare their assets and rendering techniques differently based on the blend mode they choose. For example, a black shadow rendered using the

XR_ENVIRONMENT_BLEND_MODE_ADDITIVE blend mode will appear transparent, and so an application in that mode **may** render a glow as a grounding effect around the black shadow to ensure the shadow can be seen. Similarly, an application designed for XR_ENVIRONMENT_BLEND_MODE_OPAQUE or XR_ENVIRONMENT_BLEND_MODE_ADDITIVE rendering **may** choose to leave garbage in their alpha channel as a side effect of a rendering optimization, but this garbage would appear as visible display artifacts if the environment blend mode was instead XR_ENVIRONMENT_BLEND_MODE_ALPHA_BLEND.

Not all systems will support all environment blend modes. For example, a VR headset may not support the XR_ENVIRONMENT_BLEND_MODE_ADDITIVE or XR_ENVIRONMENT_BLEND_MODE_ALPHA_BLEND modes unless it has video passthrough, while an AR headset with an additive display may not support the XR_ENVIRONMENT_BLEND_MODE_OPAQUE or XR_ENVIRONMENT_BLEND_MODE_ALPHA_BLEND modes.

For devices that can support multiple environment blend modes, such as AR phones with video passthrough, the runtime **may** optimize power consumption on the device in response to the environment blend mode that the application chooses each frame. For example, if an application on a video passthrough phone knows that it is currently rendering a 360-degree background covering all screen pixels, it can submit frames with an environment blend mode of XR_ENVIRONMENT_BLEND_MODE_OPAQUE, saving the runtime the cost of compositing a camera-based underlay of the physical world behind the application's layers.

The xrEnumerateEnvironmentBlendModes function is defined as:

```
XrResult xrEnumerateEnvironmentBlendModes(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    XrViewConfigurationType                     viewConfigurationType,
    uint32_t                                    environmentBlendModeCapacityInput,
    uint32_t*                                   environmentBlendModeCountOutput,
    XrEnvironmentBlendMode*                     environmentBlendModes);
```

Parameter Descriptions

- `instance` is the instance from which `systemId` was retrieved.
- `systemId` is the `XrSystemId` whose environment blend modes will be enumerated.
- `viewConfigurationType` is the XrViewConfigurationType to enumerate.
- `environmentBlendModeCapacityInput` is the capacity of the `environmentBlendModes` array, or 0 to indicate a request to retrieve the required capacity.
- `environmentBlendModeCountOutput` is a pointer to the count of `environmentBlendModes` written, or a pointer to the required capacity in the case that `environmentBlendModeCapacityInput` is 0.
- `environmentBlendModes` is a pointer to an array of XrEnvironmentBlendMode values, but **can** be `NULL` if `environmentBlendModeCapacityInput` is 0.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required `environmentBlendModes` size.

Enumerates the set of environment blend modes that this runtime supports for a given view configuration of the system. Environment blend modes **should** be in order from highest to lowest runtime preference.

Runtimes **must** always return identical buffer contents from this enumeration for the given `systemId` and `viewConfigurationType` for the lifetime of the instance.

Valid Usage (Implicit)

- `instance` **must** be a valid XrInstance handle
- `viewConfigurationType` **must** be a valid XrViewConfigurationType value
- `environmentBlendModeCountOutput` **must** be a pointer to a `uint32_t` value
- If `environmentBlendModeCapacityInput` is not `0`, `environmentBlendModes` **must** be a pointer to an array of `environmentBlendModeCapacityInput` XrEnvironmentBlendMode values

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_SYSTEM_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED
- XR_ERROR_SIZE_INSUFFICIENT

The possible blend modes are specified by the XrEnvironmentBlendMode enumeration:

```
typedef enum XrEnvironmentBlendMode {
    XR_ENVIRONMENT_BLEND_MODE_OPAQUE = 1,
    XR_ENVIRONMENT_BLEND_MODE_ADDITIVE = 2,
    XR_ENVIRONMENT_BLEND_MODE_ALPHA_BLEND = 3,
    XR_ENVIRONMENT_BLEND_MODE_MAX_ENUM = 0x7FFFFFFF
} XrEnvironmentBlendMode;
```

Enumerant Descriptions

- `XR_ENVIRONMENT_BLEND_MODE_OPAQUE`. The composition layers will be displayed with no view of the physical world behind them. The composited image will be interpreted as an RGB image, ignoring the composited alpha channel. This is the typical mode for VR experiences, although this mode can also be supported on devices that support video passthrough.
- `XR_ENVIRONMENT_BLEND_MODE_ADDITIVE`. The composition layers will be additively blended with the real world behind the display. The composited image will be interpreted as an RGB image, ignoring the composited alpha channel during the additive blending. This will cause black composited pixels to appear transparent. This is the typical mode for an AR experience on a see-through headset with an additive display, although this mode can also be supported on devices that support video passthrough.
- `XR_ENVIRONMENT_BLEND_MODE_ALPHA_BLEND`. The composition layers will be alpha-blended with the real world behind the display. The composited image will be interpreted as an RGBA image, with the composited alpha channel determining each pixel's level of blending with the real world behind the display. This is the typical mode for an AR experience on a phone or headset that supports video passthrough.

Chapter 11. Input and Haptics

11.1. Action Overview

OpenXR applications communicate with input devices using XrActions. Actions are created at initialization time and later used to request input device state, create action spaces, or control haptic events. Input action handles represent 'actions' that the application is interested in obtaining the state of, not direct input device hardware. For example, instead of the application directly querying the state of the A button when interacting with a menu, an OpenXR application instead creates a `menu_select` action at startup then asks OpenXR for the state of the action.

The application recommends that the action be assigned to a specific input source on the input device for a known interaction profile, but runtimes have the ability to choose a different control depending on user preference, input device availability, or any other reason. This abstraction ensures that applications can run on a wide variety of input hardware and maximize user accessibility.

Example usage:

```
XrInstance instance; // previously initialized
XrSession session; // previously initialized

// Create an action set
XrActionSetCreateInfo actionSetInfo{XR_TYPE_ACTION_SET_CREATE_INFO};
strcpy(actionSetInfo.actionSetName, "gameplay");
strcpy(actionSetInfo.localizedActionSetName, "Gameplay");
actionSetInfo.priority = 0;
XrActionSet inGameActionSet;
CHK_XR(xrCreateActionSet(instance, &actionSetInfo, &inGameActionSet));

// create a "teleport" input action
XrActionCreateInfo actioninfo{XR_TYPE_ACTION_CREATE_INFO};
strcpy(actioninfo.actionName, "teleport");
actioninfo.actionType = XR_ACTION_TYPE_BOOLEAN_INPUT;
strcpy(actioninfo.localizedActionName, "Teleport");
XrAction teleportAction;
CHK_XR(xrCreateAction(inGameActionSet, &actioninfo, &teleportAction));

// create a "player_hit" output action
XrActionCreateInfo hapticsactioninfo{XR_TYPE_ACTION_CREATE_INFO};
strcpy(hapticsactioninfo.actionName, "player_hit");
hapticsactioninfo.actionType = XR_ACTION_TYPE_VIBRATION_OUTPUT;
strcpy(hapticsactioninfo.localizedActionName, "Player hit");
XrAction hapticsAction;
CHK_XR(xrCreateAction(inGameActionSet, &hapticsactioninfo, &hapticsAction));
```

```
XrPath triggerClickPath, hapticPath;
CHK_XR(xrStringToPath(instance, "/user/hand/right/input/trigger/click",
&triggerClickPath));
CHK_XR(xrStringToPath(instance, "/user/hand/right/output/haptic", &hapticPath))

XrPath interactionProfilePath;
CHK_XR(xrStringToPath(instance, "/interaction_profiles/vendor_x/profile_x",
&interactionProfilePath));

XrActionSuggestedBinding bindings[2];
bindings[0].action = teleportAction;
bindings[0].binding = triggerClickPath;
bindings[1].action = hapticsAction;
bindings[1].binding = hapticPath;

XrInteractionProfileSuggestedBinding
suggestedBindings{XR_TYPE_INTERACTION_PROFILE_SUGGESTED_BINDING};
suggestedBindings.interactionProfile = interactionProfilePath;
suggestedBindings.suggestedBindings = bindings;
suggestedBindings.countSuggestedBindings = 2;
CHK_XR(xrSuggestInteractionProfileBindings(instance, &suggestedBindings));

XrSessionActionSetsAttachInfo attachInfo{XR_TYPE_SESSION_ACTION_SETS_ATTACH_INFO};
attachInfo.countActionSets = 1;
attachInfo.actionSets = &inGameActionSet;
CHK_XR(xrAttachSessionActionSets(session, &attachInfo));

// application main loop
while (1)
{
    // sync action data
    XrActiveActionSet activeActionSet{inGameActionSet, XR_NULL_PATH};
    XrActionsSyncInfo syncInfo{XR_TYPE_ACTIONS_SYNC_INFO};
    syncInfo.countActiveActionSets = 1;
    syncInfo.activeActionSets = &activeActionSet;
    CHK_XR(xrSyncActions(session, &syncInfo));

    // query input action state
    XrActionStateBoolean teleportState{XR_TYPE_ACTION_STATE_BOOLEAN};
    XrActionStateGetInfo getInfo{XR_TYPE_ACTION_STATE_GET_INFO};
    getInfo.action = teleportAction;
    CHK_XR(xrGetActionStateBoolean(session, &getInfo, &teleportState));

    if (teleportState.changedSinceLastSync && teleportState.currentState)
    {
        // fire haptics using output action
        XrHapticVibration vibration{XR_TYPE_HAPTIC_VIBRATION};
        vibration.amplitude = 0.5;
```

```
        vibration.duration = 300;
        vibration.frequency = 3000;
        XrHapticActionInfo hapticActionInfo{XR_TYPE_HAPTIC_ACTION_INFO};
        hapticActionInfo.action = hapticsAction;
        CHK_XR(xrApplyHapticFeedback(session, &hapticActionInfo, (const
XrHapticBaseHeader*)&vibration));
    }
}
```

11.2. Action Sets

```
XR_DEFINE_HANDLE(XrActionSet)
```

Action sets are application-defined collections of actions. They are attached to a given XrSession with a xrAttachSessionActionSets call. They are enabled or disabled by the application via xrSyncActions depending on the current application context. For example, a game may have one set of actions that apply to controlling a character and another set for navigating a menu system. When these actions are grouped into two XrActionSet handles they can be selectively enabled and disabled using a single function call.

Actions are passed a handle to their XrActionSet when they are created.

Action sets are created by calling xrCreateActionSet:

The xrCreateActionSet function is defined as:

```
XrResult xrCreateActionSet(
    XrInstance                                  instance,
    const XrActionSetCreateInfo*                createInfo,
    XrActionSet*                                actionSet);
```

Parameter Descriptions

- `instance` is a handle to an XrInstance.
- `createInfo` is a pointer to a valid XrActionSetCreateInfo structure that defines the action set being created.
- `actionSet` is a pointer to an XrActionSet where the created action set is returned.

The xrCreateActionSet function creates an action set and returns a handle to the created action set.

Valid Usage (Implicit)

- `instance` **must** be a valid XrInstance handle
- `createInfo` **must** be a pointer to a valid XrActionSetCreateInfo structure
- `actionSet` **must** be a pointer to an XrActionSet handle

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_LIMIT_REACHED`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_OUT_OF_MEMORY`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_NAME_DUPLICATED`
- `XR_ERROR_LOCALIZED_NAME_DUPLICATED`
- `XR_ERROR_NAME_INVALID`
- `XR_ERROR_LOCALIZED_NAME_INVALID`
- `XR_ERROR_PATH_FORMAT_INVALID`

The XrActionSetCreateInfo structure is defined as:

```
typedef struct XrActionSetCreateInfo {
    XrStructureType    type;
    const void*        next;
    char               actionSetName[XR_MAX_ACTION_SET_NAME_SIZE];
    char               localizedActionSetName[XR_MAX_LOCALIZED_ACTION_SET_NAME_SIZE];
    uint32_t           priority;
} XrActionSetCreateInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `actionSetName` is an array containing a `NULL` terminated non-empty string with the name of this action set.
- `localizedActionSetName` is an array containing a `NULL` terminated `UTF`-8 string that can be presented to the user as a description of the action set. This string should be presented in the system's current active locale.
- `priority` defines which action sets' actions are active on a given input source when actions on multiple active action sets are bound to the same input source. Larger priority numbers take precedence over smaller priority numbers.

When multiple actions are bound to the same input source, the `priority` of each action set determines which bindings are suppressed. Runtimes **must** ignore input sources from action sets with a lower priority number if those specific input sources are also present in active actions within a higher priority action set. If multiple action sets with the same priority are bound to the same input source and that is the highest priority number, runtimes **must** process all those bindings at the same time.

Two actions are considered to be bound to the same input source if they use the same identifier and optional location path segments, even if they have different component segments.

When runtimes are ignoring bindings because of priority, they **must** treat the binding to that input source as though they do not exist. That means the `isActive` field **must** be `XR_FALSE` when retrieving action data, and that the runtime **must** not provide any visual, haptic, or other feedback related to the binding of that action to that input source. Other actions in the same action set which are bound to input sources that do not collide are not affected and are processed as normal.

If `actionSetName` or `localizedActionSetName` are empty strings, the runtime **must** return `XR_ERROR_NAME_INVALID` or `XR_ERROR_LOCALIZED_NAME_INVALID` respectively. If `actionSetName` or `localizedActionSetName` are duplicates of the corresponding field for any existing action set in the specified instance, the runtime **must** return `XR_ERROR_NAME_DUPLICATED` or `XR_ERROR_LOCALIZED_NAME_DUPLICATED` respectively. If the conflicting action set is destroyed, the conflicting field is no longer considered duplicated. If `actionSetName` contains characters which are not allowed in a single level of a well-formed path string, the runtime **must** return `XR_ERROR_PATH_FORMAT_INVALID`.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_ACTION_SET_CREATE_INFO`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `actionSetName` **must** be a null-terminated UTF-8 string whose length is less than or equal to `XR_MAX_ACTION_SET_NAME_SIZE`
- `localizedActionSetName` **must** be a null-terminated UTF-8 string whose length is less than or equal to `XR_MAX_LOCALIZED_ACTION_SET_NAME_SIZE`

The xrDestroyActionSet function is defined as:

```
XrResult xrDestroyActionSet(
    XrActionSet                                 actionSet);
```

Parameter Descriptions

- `actionSet` is the action set to destroy.

Action set handles **can** be destroyed by calling xrDestroyActionSet. When an action set handle is destroyed, all handles of actions in that action set are also destroyed.

The implementation **must** not free underlying resources for the action set while there are other valid handles that refer to those resources. The implementation **may** release resources for an action set when all of the action spaces for actions in that action set have been destroyed. See Action Spaces Lifetime for details.

Resources for all action sets in an instance **must** be freed when the instance containing those actions sets is destroyed.

Valid Usage (Implicit)

- `actionSet` **must** be a valid XrActionSet handle

Thread Safety

- Access to `actionSet`, and any child handles, **must** be externally synchronized

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID

11.3. Creating Actions

```
XR_DEFINE_HANDLE(XrAction)
```

Action handles are used to refer to individual actions when retrieving action data, creating action spaces, or sending haptic events.

The xrCreateAction function is defined as:

```
XrResult xrCreateAction(
    XrActionSet                                 actionSet,
    const XrActionCreateInfo*                   createInfo,
    XrAction*                                   action);
```

Parameter Descriptions

- actionSet is a handle to an XrActionSet.
- createInfo is a pointer to a valid XrActionCreateInfo structure that defines the action being created.
- action is a pointer to an XrAction where the created action is returned.

xrCreateAction creates an action and returns its handle.

If actionSet has been included in a call to xrAttachSessionActionSets, the implementation **must** return XR_ERROR_ACTIONSETS_ALREADY_ATTACHED.

Valid Usage (Implicit)

- `actionSet` **must** be a valid XrActionSet handle
- `createInfo` **must** be a pointer to a valid XrActionCreateInfo structure
- `action` **must** be a pointer to an XrAction handle

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_ACTIONSETS_ALREADY_ATTACHED`
- `XR_ERROR_LIMIT_REACHED`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_OUT_OF_MEMORY`
- `XR_ERROR_PATH_INVALID`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_NAME_DUPLICATED`
- `XR_ERROR_LOCALIZED_NAME_DUPLICATED`
- `XR_ERROR_NAME_INVALID`
- `XR_ERROR_LOCALIZED_NAME_INVALID`
- `XR_ERROR_PATH_FORMAT_INVALID`
- `XR_ERROR_PATH_UNSUPPORTED`

The XrActionCreateInfo structure is defined as:

```
typedef struct XrActionCreateInfo {
    XrStructureType    type;
    const void*        next;
    char               actionName[XR_MAX_ACTION_NAME_SIZE];
    XrActionType       actionType;
    uint32_t           countSubactionPaths;
    const XrPath*      subactionPaths;
    char               localizedActionName[XR_MAX_LOCALIZED_ACTION_NAME_SIZE];
} XrActionCreateInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `actionName` is an array containing a `NULL` terminated string with the name of this action.
- `actionType` is the XrActionType of the action to be created.
- `countSubactionPaths` is the number of elements in the `subactionPaths` array. If `subactionPaths` is NULL, this parameter must be 0.
- `subactionPaths` is an array of `XrPath` or `NULL`. If this array is specified, it contains one or more subaction paths that the application intends to query action state for.
- `localizedActionName` is an array containing a `NULL` terminated `UTF`-8 string that can be presented to the user as a description of the action. This string should be in the system's current active locale.

Subaction paths are a mechanism that enables applications to use the same action name and handle on multiple devices. Applications can query action state using subaction paths that differentiate data coming from each device. This allows the runtime to group logically equivalent actions together in system UI. For instance, an application could create a single `pick_up` action with the */user/hand/left* and */user/hand/right* subaction paths and use the subaction paths to independently query the state of `pick_up_with_left_hand` and `pick_up_with_right_hand`.

Applications **can** create actions with or without the `subactionPaths` set to a list of paths. If this list of paths is omitted (i.e. `subactionPaths` is set to `NULL`, and `countSubactionPaths` is set to `0`), the application is opting out of filtering action results by subaction paths and any call to get action data must also omit subaction paths.

If `subactionPaths` is specified and any of the following conditions are not satisfied, the runtime **must** return `XR_ERROR_PATH_UNSUPPORTED`:

- Each path provided is one of:

    - */user/head*
    - */user/hand/left*
    - */user/hand/right*
    - */user/gamepad*
- No path appears in the list more than once

Extensions **may** append additional top level user paths to the above list.

> *Note*
>
> Earlier revisions of the spec mentioned */user* but it could not be implemented as specified and was removed as errata.

The runtime **must** return `XR_ERROR_PATH_UNSUPPORTED` in the following circumstances:

- The application specified subaction paths at action creation and the application called `xrGetActionState*` or a haptic function with an empty subaction path array.
- The application called `xrGetActionState*` or a haptic function with a subaction path that was not specified when the action was created.

If `actionName` or `localizedActionName` are empty strings, the runtime **must** return `XR_ERROR_NAME_INVALID` or `XR_ERROR_LOCALIZED_NAME_INVALID` respectively. If `actionName` or `localizedActionName` are duplicates of the corresponding field for any existing action in the specified action set, the runtime **must** return `XR_ERROR_NAME_DUPLICATED` or `XR_ERROR_LOCALIZED_NAME_DUPLICATED` respectively. If the conflicting action is destroyed, the conflicting field is no longer considered duplicated. If `actionName` contains characters which are not allowed in a single level of a well-formed path string, the runtime **must** return `XR_ERROR_PATH_FORMAT_INVALID`.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_ACTION_CREATE_INFO`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `actionName` **must** be a null-terminated UTF-8 string whose length is less than or equal to `XR_MAX_ACTION_NAME_SIZE`
- `actionType` **must** be a valid XrActionType value
- If `countSubactionPaths` is not `0`, `subactionPaths` **must** be a pointer to an array of `countSubactionPaths` valid `XrPath` values
- `localizedActionName` **must** be a null-terminated UTF-8 string whose length is less than or equal to `XR_MAX_LOCALIZED_ACTION_NAME_SIZE`

The XrActionType parameter takes one of the following values:

```
typedef enum XrActionType {
    XR_ACTION_TYPE_BOOLEAN_INPUT = 1,
    XR_ACTION_TYPE_FLOAT_INPUT = 2,
    XR_ACTION_TYPE_VECTOR2F_INPUT = 3,
    XR_ACTION_TYPE_POSE_INPUT = 4,
    XR_ACTION_TYPE_VIBRATION_OUTPUT = 100,
    XR_ACTION_TYPE_MAX_ENUM = 0x7FFFFFFF
} XrActionType;
```

Enumerant Descriptions

- `XR_ACTION_TYPE_BOOLEAN_INPUT`. The action can be passed to xrGetActionStateBoolean to retrieve a boolean value.
- `XR_ACTION_TYPE_FLOAT_INPUT`. The action can be passed to xrGetActionStateFloat to retrieve a float value.
- `XR_ACTION_TYPE_VECTOR2F_INPUT`. The action can be passed to xrGetActionStateVector2f to retrieve a 2D float vector.
- `XR_ACTION_TYPE_POSE_INPUT`. The action can can be passed to xrCreateActionSpace to create a space.
- `XR_ACTION_TYPE_VIBRATION_OUTPUT`. The action can be passed to xrApplyHapticFeedback to send a haptic event to the runtime.

The xrDestroyAction function is defined as:

```
XrResult xrDestroyAction(
    XrAction                                    action);
```

Parameter Descriptions

- `action` is the action to destroy.

Action handles **can** be destroyed by calling xrDestroyAction. Handles for actions that are part of an action set are automatically destroyed when the action set's handle is destroyed.

The implementation **must** not destroy the underlying resources for an action when xrDestroyAction is called. Those resources are still used to make action spaces locatable and when processing action priority in xrSyncActions. Destroying the action handle removes the application's access to these resources, but has no other change on actions.

Resources for all actions in an instance **must** be freed when the instance containing those actions sets is destroyed.

Valid Usage (Implicit)

- `action` **must** be a valid XrAction handle

Thread Safety

- Access to `action`, and any child handles, **must** be externally synchronized

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_HANDLE_INVALID`

11.3.1. Input Actions & Output Actions

Input actions are used to read sensors like buttons or joysticks while output actions are used for triggering haptics or motion platforms. The type of action created by xrCreateAction depends on the value of the XrActionType argument.

A given action can either be used for either input or output, but not both. Input actions are queried using one of the `xrGetActionState*` function calls, while output actions are set using the haptics calls. If either call is used with an action of the wrong type `XR_ERROR_ACTION_TYPE_MISMATCH` **must** be returned.

11.4. Suggested Bindings

Applications usually need to provide default bindings for their actions to runtimes so that input data can be mapped appropriately to the application's actions. Applications **can** do this by calling xrSuggestInteractionProfileBindings for each interaction profile that the applications has default bindings for. If bindings are provided for an appropriate interaction profile, the runtime **may** select one and input will begin to flow. Interaction profile selection changes **must** only happen when xrSyncActions is called. Applications **can** call xrGetCurrentInteractionProfile during on a running session to learn what the active interaction profile are for a top level user path. If this value ever changes, the runtime **must** send an `XR_TYPE_EVENT_DATA_INTERACTION_PROFILE_CHANGED` event to the application to indicate that the value should be queried again.

The bindings suggested by this system are only a hint to the runtime. Some runtimes **may** choose to use a different device binding depending on user preference, accessibility settings, or for any other reason. If the runtime is using the values provided by suggested bindings, it **must** make a best effort to convert the input value to the created action and apply certain rules to that use so that suggested bindings function in the same way across runtimes. If an input value cannot be converted to the type of the action, the value **must** be ignored and not contribute to the state of the action.

For actions created with `XR_ACTION_TYPE_BOOLEAN_INPUT` when the runtime is obeying suggested bindings: Boolean input sources **must** be bound directly to the action. If the path is to a scalar value, a threshold **must** be applied to the value and values over that threshold will be `XR_TRUE`. The runtime **should** use hysteresis when applying this threshold. The threshold and hysteresis range **may** vary from device to device or component to component and are left as an implementation detail. If the path refers to the parent of input values instead of to an input value itself, the runtime **must** use *…/example/path/value* instead of *…/example/path* if it is available and apply the same thresholding that would be applied to any scalar input. If a parent path does not have a *…/value* subpath, the runtime **must** use *…/click*. In any other situation the runtime **may** provide an alternate binding for the action or it will be unbound.

For actions created with `XR_ACTION_TYPE_FLOAT_INPUT` when the runtime is obeying suggested bindings: If the input value specified by the path is scalar, the input value **must** be bound directly to the float. If the path refers to the parent of input values instead of to an input value itself, the runtime **must** use */example/path/value* instead of *…/example/path* as the source of the value. If the input value is boolean, the runtime **must** supply 0.0 or 1.0 as a conversion of the boolean value. In any other situation, the runtime **may** provide an alternate binding for the action or it will be unbound.

For actions created with `XR_ACTION_TYPE_VECTOR2F_INPUT` when the runtime is obeying suggested bindings: The suggested binding path **must** refer to the parent of input values instead of to the input values themselves, and that parent path **must** contain subpaths *…/x* and *…/y*. *…/x* and *…/y* **must** be bound to 'x' and 'y' of the vector, respectively. In any other situation, the runtime **may** provide an alternate binding for the action or it will be unbound.

For actions created with `XR_ACTION_TYPE_POSE_INPUT` when the runtime is obeying suggested bindings: Pose input sources **must** be bound directly to the action. If the path refers to the parent of input values instead of to an input value itself, the runtime **must** use *…/example/path/pose* instead of *…/example/path* if it is available. In any other situation the runtime **may** provide an alternate binding for the action or it will be unbound.

The XrEventDataInteractionProfileChanged structure is defined as:

```
typedef struct XrEventDataInteractionProfileChanged {
    XrStructureType    type;
    const void*        next;
    XrSession          session;
} XrEventDataInteractionProfileChanged;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `session` is the XrSession for which at least one of the interaction profiles for a top level path has changed.

The XrEventDataInteractionProfileChanged event is sent to the application to notify it that the active input form factor for one or more top level user paths has changed. This event **must** only be sent for interaction profiles that the application indicated its support for via xrSuggestInteractionProfileBindings. This event **must** only be sent for running sessions.

The application **can** call xrGetCurrentInteractionProfile if it wants to change its own behavior based on the active hardware.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_EVENT_DATA_INTERACTION_PROFILE_CHANGED`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `session` **must** be a valid XrSession handle

The xrSuggestInteractionProfileBindings function is defined as:

```
XrResult xrSuggestInteractionProfileBindings(
    XrInstance                                  instance,
    const XrInteractionProfileSuggestedBinding* suggestedBindings);
```

Parameter Descriptions

- `instance` is the XrInstance for which the application would like to set suggested bindings
- `suggestedBindings` is the XrInteractionProfileSuggestedBinding that the application would like to set

xrSuggestInteractionProfileBindings sets an interaction profile for which the application can provide default bindings. The application **can** call xrSuggestInteractionProfileBindings once per interaction profile that it supports.

The application **can** provide any number of bindings for each action.

If the application successfully calls xrSuggestInteractionProfileBindings more than once for an interaction profile, the runtime **must** discard the previous suggested bindings and replace them with the new suggested bindings for that profile.

If the interaction profile path does not follow the structure defined in Interaction Profiles or suggested bindings contain paths that do not follow the format defined in Device input subpaths, the runtime **must** return `XR_ERROR_PATH_UNSUPPORTED`. If the interaction profile or input source for any of the suggested bindings does not exist in the allowlist defined in Interaction Profile Paths, the runtime **must** return `XR_ERROR_PATH_UNSUPPORTED`. A runtime **must** accept every valid binding in the allowlist though it is free to ignore any of them.

If the action set for any action referenced in the `suggestedBindings` parameter has been included in a call to xrAttachSessionActionSets, the implementation **must** return `XR_ERROR_ACTIONSETS_ALREADY_ATTACHED`.

Valid Usage (Implicit)

- `instance` **must** be a valid XrInstance handle
- `suggestedBindings` **must** be a pointer to a valid XrInteractionProfileSuggestedBinding structure

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_ACTIONSETS_ALREADY_ATTACHED
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_PATH_UNSUPPORTED
- XR_ERROR_PATH_INVALID

The XrInteractionProfileSuggestedBinding structure is defined as:

```
typedef struct XrInteractionProfileSuggestedBinding {
    XrStructureType                     type;
    const void*                         next;
    XrPath                              interactionProfile;
    uint32_t                            countSuggestedBindings;
    const XrActionSuggestedBinding*     suggestedBindings;
} XrInteractionProfileSuggestedBinding;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `interactionProfile` is the `XrPath` of an interaction profile.
- `countSuggestedBindings` is the number of suggested bindings in the array pointed to by `suggestedBindings`.
- `suggestedBindings` is a pointer to an array of XrActionSuggestedBinding structures that define all of the application's suggested bindings for the specified interaction profile.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_INTERACTION_PROFILE_SUGGESTED_BINDING`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrBindingModificationsKHR, XrInteractionProfileAnalogThresholdVALVE
- `suggestedBindings` **must** be a pointer to an array of `countSuggestedBindings` valid XrActionSuggestedBinding structures
- The `countSuggestedBindings` parameter **must** be greater than `0`

The XrActionSuggestedBinding structure is defined as:

```
typedef struct XrActionSuggestedBinding {
    XrAction    action;
    XrPath      binding;
} XrActionSuggestedBinding;
```

Member Descriptions

- `action` is the XrAction handle for an action
- `binding` is the `XrPath` of a binding for the action specified in `action`. This path is any top level user path plus input source path, for example */user/hand/right/input/trigger/click*. See suggested bindings for more details.

Valid Usage (Implicit)

- `action` **must** be a valid XrAction handle

The xrAttachSessionActionSets function is defined as:

```
XrResult xrAttachSessionActionSets(
    XrSession                                   session,
    const XrSessionActionSetsAttachInfo*        attachInfo);
```

Parameter Descriptions

- `session` is the XrSession to attach the action sets to.
- `attachInfo` is the XrSessionActionSetsAttachInfo to provide information to attach action sets to the session.

xrAttachSessionActionSets attaches the XrActionSet handles in `attachInfo.actionSets` to the `session`. Action sets **must** be attached in order to be synchronized with xrSyncActions.

When an action set is attached to a session, that action set becomes immutable. See xrCreateAction and xrSuggestInteractionProfileBindings for details.

After action sets are attached to a session, if any unattached actions are passed to functions for the same session, then for those functions the runtime **must** return `XR_ERROR_ACTIONSET_NOT_ATTACHED`.

The runtime **must** return `XR_ERROR_ACTIONSETS_ALREADY_ATTACHED` if xrAttachSessionActionSets is called more than once for a given `session`.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- `attachInfo` **must** be a pointer to a valid XrSessionActionSetsAttachInfo structure

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_ACTIONSETS_ALREADY_ATTACHED`

The XrSessionActionSetsAttachInfo structure is defined as:

```
typedef struct XrSessionActionSetsAttachInfo {
    XrStructureType       type;
    const void*           next;
    uint32_t              countActionSets;
    const XrActionSet*    actionSets;
} XrSessionActionSetsAttachInfo;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- countActionSets is an integer specifying the number of valid elements in the actionSets array.
- actionSets is a pointer to an array of one or more XrActionSet handles to be attached to the session.

Valid Usage (Implicit)

- type **must** be XR_TYPE_SESSION_ACTION_SETS_ATTACH_INFO
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- actionSets **must** be a pointer to an array of countActionSets valid XrActionSet handles
- The countActionSets parameter **must** be greater than 0

The xrGetCurrentInteractionProfile function is defined as:

```
XrResult xrGetCurrentInteractionProfile(
    XrSession                                   session,
    XrPath                                      topLevelUserPath,
    XrInteractionProfileState*                  interactionProfile);
```

Parameter Descriptions

- `session` is the XrSession for which the application would like to retrieve the current interaction profile.
- `topLevelUserPath` is the top level user path the application would like to retrieve the interaction profile for.
- `interactionProfile` is a pointer to an XrInteractionProfileState structure to receive the current interaction profile.

xrGetCurrentInteractionProfile asks the runtime for the active interaction profiles for a top level user path.

The runtime **must** return only interaction profiles for which the application has provided bindings with xrSuggestInteractionProfileBindings or XR_NULL_PATH. The runtime **may** return interaction profiles that do not represent physically present hardware, for example if the runtime is using a known interaction profile to bind to hardware that the application is not aware of. The runtime **may** return the last-known interaction profile in the event that no controllers are active.

If xrAttachSessionActionSets has not yet been called for the `session`, the runtime **must** return `XR_ERROR_ACTIONSET_NOT_ATTACHED`. If `topLevelUserPath` is not one of the device input subpaths described in section /user paths, the runtime **must** return `XR_ERROR_PATH_UNSUPPORTED`.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- `interactionProfile` **must** be a pointer to an XrInteractionProfileState structure

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_ACTIONSET_NOT_ATTACHED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_PATH_UNSUPPORTED
- XR_ERROR_PATH_INVALID

The XrInteractionProfileState structure is defined as:

```
typedef struct XrInteractionProfileState {
    XrStructureType    type;
    void*              next;
    XrPath             interactionProfile;
} XrInteractionProfileState;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- interactionProfile is the XrPath of the interaction profile path for the topLevelUserPath used to retrieve this state, or XR_NULL_PATH if there is no active interaction profile at that top level user path.

The runtime **must** only include interaction profiles that the application has provided bindings for via xrSuggestInteractionProfileBindings or XR_NULL_PATH. If the runtime is rebinding an interaction profile provided by the application to a device that the application did not provide bindings for, it

must return the interaction profile path that it is emulating. If the runtime is unable to provide input because it cannot emulate any of the application-provided interaction profiles, it **must** return XR_NULL_PATH.

> **Valid Usage (Implicit)**
>
> - `type` **must** be `XR_TYPE_INTERACTION_PROFILE_STATE`
> - `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

11.5. Reading Input Action State

The current state of an input action can be obtained by calling the `xrGetActionState*` function call that matches the XrActionType provided when the action was created. If a mismatched call is used to retrieve the state `XR_ERROR_ACTION_TYPE_MISMATCH` **must** be returned. `xrGetActionState*` calls for an action in an action set never bound to the session with xrAttachSessionActionSets **must** return `XR_ERROR_ACTIONSET_NOT_ATTACHED`.

The result of calls to `xrGetActionState*` for an XrAction and subaction path **must** not change between calls to xrSyncActions. When the combination of the parent XrActionSet and subaction path for an action is passed to xrSyncActions, the runtime **must** update the results from `xrGetActionState*` after this call with any changes to the state of the underlying hardware. When the parent action set and subaction path for an action is removed from or added to the list of active action sets passed to xrSyncActions, the runtime **must** update `isActive` to reflect the new active state after this call. In all cases the runtime **must** not change the results of `xrGetActionState*` calls between calls to xrSyncActions.

When `xrGetActionState*` or haptic output functions are called while the session is not running, the runtime **must** set the `isActive` value to `XR_FALSE` and suppress all haptic output.

When retrieving action state, `lastChangeTime` **must** be set to the runtime's best estimate of when the physical state of the part of the device bound to that action last changed.

The `currentState` value is computed based on the current sync, combining the underlying input sources bound to the provided `subactionPaths` within this action.

The `changedSinceLastSync` value **must** be `XR_TRUE` if the computed `currentState` value differs from the `currentState` value that would have been computed as of the previous sync for the same `subactionPaths`. If there is no previous sync, or the action was not active for the previous sync, the `changedSinceLastSync` value **must** be set to `XR_FALSE`.

The `isActive` value **must** be `XR_TRUE` whenever an action is bound and a source is providing state data for the current sync. If the action is unbound or no source is present, the `isActive` value **must** be `XR_FALSE`. For any action which is inactive, the runtime **must** return zero (or `XR_FALSE`) for state, `XR_FALSE` for `changedSinceLastSync`, and `0` for `lastChangeTime`.

11.5.1. Resolving a single action bound to multiple inputs or outputs

It is often the case that a single action will be bound to multiple physical inputs simultaneously. In these circumstances, the runtime **must** resolve the ambiguity in that multiple binding as follows:

The current state value is selected based on the type of the action:

- Boolean actions - The current state **must** be the result of a boolean OR of all bound inputs
- Float actions - The current state **must** be the state of the input with the largest absolute value
- Vector2 actions - The current state **must** be the state of the input with the longest length
- Pose actions - The runtime **must** select a single pose source when the action is created or bound and use that value consistently. The runtime **should** use subaction paths specified by the application to make this choice where possible.
- Haptic actions - The runtime **must** send output events to all bound haptic devices

11.5.2. Structs to describe action and subaction paths

The XrActionStateGetInfo structure is used to provide action and subaction paths when calling xrGetActionState* function. It is defined as:

```
typedef struct XrActionStateGetInfo {
    XrStructureType    type;
    const void*        next;
    XrAction           action;
    XrPath             subactionPath;
} XrActionStateGetInfo;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- action is the XrAction being queried.
- subactionPath is the subaction path XrPath to query data from, or XR_NULL_PATH to specify all subaction paths. If the subaction path is specified, it is one of the subaction paths that were specified when the action was created. If the subaction path was not specified when the action was created, the runtime **must** return XR_ERROR_PATH_UNSUPPORTED. If this parameter is specified, the runtime **must** return data that originates only from the subaction paths specified.

See XrActionCreateInfo for a description of subaction paths, and the restrictions on their use.

Valid Usage (Implicit)

- type **must** be XR_TYPE_ACTION_STATE_GET_INFO
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- action **must** be a valid XrAction handle

The XrHapticActionInfo structure is used to provide action and subaction paths when calling xr*HapticFeedback function. It is defined as:

```
typedef struct XrHapticActionInfo {
    XrStructureType    type;
    const void*        next;
    XrAction           action;
    XrPath             subactionPath;
} XrHapticActionInfo;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- action is the XrAction handle for the desired output haptic action.
- subactionPath is the subaction path XrPath of the device to send the haptic event to, or XR_NULL_PATH to specify all subaction paths. If the subaction path is specified, it is one of the subaction paths that were specified when the action was created. If the subaction path was not specified when the action was created, the runtime **must** return XR_ERROR_PATH_UNSUPPORTED. If this parameter is specified, the runtime **must** trigger the haptic events only on the device from the subaction path.

See XrActionCreateInfo for a description of subaction paths, and the restrictions on their use.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_HAPTIC_ACTION_INFO`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `action` **must** be a valid XrAction handle

11.5.3. Boolean Actions

xrGetActionStateBoolean retrieves the current state of a boolean action. It is defined as:

```
XrResult xrGetActionStateBoolean(
    XrSession                                   session,
    const XrActionStateGetInfo*                 getInfo,
    XrActionStateBoolean*                       state);
```

Parameter Descriptions

- `session` is the XrSession to query.
- `getInfo` is a pointer to XrActionStateGetInfo to provide action and subaction paths information.
- `state` is a pointer to a valid XrActionStateBoolean into which the state will be placed.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- `getInfo` **must** be a pointer to a valid XrActionStateGetInfo structure
- `state` **must** be a pointer to an XrActionStateBoolean structure

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_ACTIONSET_NOT_ATTACHED
- XR_ERROR_ACTION_TYPE_MISMATCH
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_PATH_INVALID
- XR_ERROR_PATH_UNSUPPORTED

The XrActionStateBoolean structure is defined as:

```
typedef struct XrActionStateBoolean {
    XrStructureType    type;
    void*              next;
    XrBool32           currentState;
    XrBool32           changedSinceLastSync;
    XrTime             lastChangeTime;
    XrBool32           isActive;
} XrActionStateBoolean;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- currentState is the current state of the action.
- changedSinceLastSync is XR_TRUE if the value of currentState is different than it was before the most recent call to xrSyncActions. This parameter can be combined with currentState to detect rising and falling edges since the previous call to xrSyncActions. E.g. if both changedSinceLastSync and currentState are XR_TRUE then a rising edge (XR_FALSE to XR_TRUE) has taken place.
- lastChangeTime is the XrTime when this action's value last changed.
- isActive is XR_TRUE if and only if there exists an input source that is contributing to the current state of this action.

When multiple input sources are bound to this action, the currentState follows the previously defined rule to resolve ambiguity.

Valid Usage (Implicit)

- type **must** be XR_TYPE_ACTION_STATE_BOOLEAN
- next **must** be NULL or a valid pointer to the next structure in a structure chain

11.5.4. Scalar and Vector Actions

xrGetActionStateFloat retrieves the current state of a floating-point action. It is defined as:

```
XrResult xrGetActionStateFloat(
    XrSession                                   session,
    const XrActionStateGetInfo*                 getInfo,
    XrActionStateFloat*                         state);
```

Parameter Descriptions

- `session` is the XrSession to query.
- `getInfo` is a pointer to XrActionStateGetInfo to provide action and subaction paths information.
- `state` is a pointer to a valid XrActionStateFloat into which the state will be placed.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- `getInfo` **must** be a pointer to a valid XrActionStateGetInfo structure
- `state` **must** be a pointer to an XrActionStateFloat structure

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_ACTIONSET_NOT_ATTACHED`
- `XR_ERROR_ACTION_TYPE_MISMATCH`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_PATH_INVALID`
- `XR_ERROR_PATH_UNSUPPORTED`

The XrActionStateFloat structure is defined as:

```
typedef struct XrActionStateFloat {
    XrStructureType    type;
    void*              next;
    float              currentState;
    XrBool32           changedSinceLastSync;
    XrTime             lastChangeTime;
    XrBool32           isActive;
} XrActionStateFloat;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `currentState` is the current state of the Action.
- `changedSinceLastSync` is `XR_TRUE` if the value of `currentState` is different than it was before the most recent call to xrSyncActions.
- `lastChangeTime` is the `XrTime` in nanoseconds since this action's value last changed.
- `isActive` is `XR_TRUE` if and only if there exists an input source that is contributing to the current state of this action.

When multiple input sources are bound to this action, the `currentState` follows the previously defined rule to resolve ambiguity.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_ACTION_STATE_FLOAT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

xrGetActionStateVector2f retrieves the current state of a two-dimensional vector action. It is defined as:

```
XrResult xrGetActionStateVector2f(
    XrSession                                   session,
    const XrActionStateGetInfo*                 getInfo,
    XrActionStateVector2f*                      state);
```

Parameter Descriptions

- `session` is the XrSession to query.
- `getInfo` is a pointer to XrActionStateGetInfo to provide action and subaction paths information.
- `state` is a pointer to a valid XrActionStateVector2f into which the state will be placed.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- `getInfo` **must** be a pointer to a valid XrActionStateGetInfo structure
- `state` **must** be a pointer to an XrActionStateVector2f structure

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_ACTIONSET_NOT_ATTACHED`
- `XR_ERROR_ACTION_TYPE_MISMATCH`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_PATH_INVALID`
- `XR_ERROR_PATH_UNSUPPORTED`

The XrActionStateVector2f structure is defined as:

```
typedef struct XrActionStateVector2f {
    XrStructureType    type;
    void*              next;
    XrVector2f         currentState;
    XrBool32           changedSinceLastSync;
    XrTime             lastChangeTime;
    XrBool32           isActive;
} XrActionStateVector2f;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `currentState` is the current XrVector2f state of the Action.
- `changedSinceLastSync` is `XR_TRUE` if the value of `currentState` is different than it was before the most recent call to xrSyncActions.
- `lastChangeTime` is the `XrTime` in nanoseconds since this action's value last changed.
- `isActive` is `XR_TRUE` if and only if there exists an input source that is contributing to the current state of this action.

When multiple input sources are bound to this action, the `currentState` follows the previously defined rule to resolve ambiguity.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_ACTION_STATE_VECTOR2F`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

11.5.5. Pose Actions

The xrGetActionStatePose function is defined as:

```
XrResult xrGetActionStatePose(
    XrSession                                   session,
    const XrActionStateGetInfo*                 getInfo,
    XrActionStatePose*                          state);
```

Parameter Descriptions

- `session` is the XrSession to query.
- `getInfo` is a pointer to XrActionStateGetInfo to provide action and subaction paths information.
- `state` is a pointer to a valid XrActionStatePose into which the state will be placed.

xrGetActionStatePose returns information about the binding and active state for the specified action. To determine the pose of this action at a historical or predicted time, the application **can** create an action space using xrCreateActionSpace. Then, after each sync, the application **can** locate the pose of this action space within a base space using xrLocateSpace.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- `getInfo` **must** be a pointer to a valid XrActionStateGetInfo structure
- `state` **must** be a pointer to an XrActionStatePose structure

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_ACTIONSET_NOT_ATTACHED`
- `XR_ERROR_ACTION_TYPE_MISMATCH`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_PATH_INVALID`
- `XR_ERROR_PATH_UNSUPPORTED`

The XrActionStatePose structure is defined as:

```
typedef struct XrActionStatePose {
    XrStructureType    type;
    void*              next;
    XrBool32           isActive;
} XrActionStatePose;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `isActive` is `XR_TRUE` if and only if there exists an input source that is being tracked by this pose action.

A pose action **must** not be bound to multiple input sources, according to the previously defined rule.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_ACTION_STATE_POSE`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

11.6. Output Actions and Haptics

Haptic feedback is sent to a device using the xrApplyHapticFeedback function. The `hapticEvent` points to a supported event structure. All event structures have in common that the first element is an XrHapticBaseHeader which can be used to determine the type of the haptic event.

Haptic feedback may be immediately halted for a haptic action using the xrStopHapticFeedback function.

Output action requests activate immediately and **must** not wait for the next call to xrSyncActions.

If a haptic event is sent to an action before a previous haptic event completes, the latest event will take precedence and the runtime **must** cancel all preceding incomplete haptic events on that action.

Output action requests **must** be discarded and have no effect on hardware if the application's session is not active.

Output action requests for an action in an action set never attached to the session with xrAttachSessionActionSets **must** return XR_ERROR_ACTIONSET_NOT_ATTACHED.

The only haptics type supported by unextended OpenXR is XrHapticVibration.

The xrApplyHapticFeedback function is defined as:

```
XrResult xrApplyHapticFeedback(
    XrSession                                   session,
    const XrHapticActionInfo*                   hapticActionInfo,
    const XrHapticBaseHeader*                   hapticFeedback);
```

Parameter Descriptions

- session is the XrSession to start outputting to.
- hapticActionInfo is a pointer to XrHapticActionInfo to provide action and subaction paths information.
- hapticFeedback is a pointer to a haptic event structure which starts with an XrHapticBaseHeader.

Triggers a haptic event through the specified action of type XR_TYPE_HAPTIC_VIBRATION. The runtime **should** deliver this request to the appropriate device, but exactly which device, if any, this event is sent to is up to the runtime to decide. If an appropriate device is unavailable the runtime **may** ignore this request for haptic feedback.

If another haptic event from this session is currently happening on the device bound to this action, the runtime **must** interrupt that other event and replace it with the new one.

Valid Usage (Implicit)

- session **must** be a valid XrSession handle
- hapticActionInfo **must** be a pointer to a valid XrHapticActionInfo structure
- hapticFeedback **must** be a pointer to a valid XrHapticBaseHeader-based structure. See also: XrHapticVibration

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_ACTIONSET_NOT_ATTACHED
- XR_ERROR_ACTION_TYPE_MISMATCH
- XR_ERROR_PATH_INVALID
- XR_ERROR_PATH_UNSUPPORTED

The XrHapticBaseHeader structure is defined as:

```
typedef struct XrHapticBaseHeader {
    XrStructureType    type;
    const void*        next;
} XrHapticBaseHeader;
```

Member Descriptions

- type is the XrStructureType of this structure. This base structure itself has no associated XrStructureType value.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_HAPTIC_VIBRATION`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

The XrHapticVibration structure is defined as:

```
typedef struct XrHapticVibration {
    XrStructureType    type;
    const void*        next;
    XrDuration         duration;
    float              frequency;
    float              amplitude;
} XrHapticVibration;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `duration` is the number of nanoseconds the vibration **should** last. If XR_MIN_HAPTIC_DURATION is specified, the runtime **must** produce a short haptics pulse of minimal supported duration for the haptic device.
- `frequency` is the frequency of the vibration in Hz. If XR_FREQUENCY_UNSPECIFIED is specified, it is left to the runtime to decide the optimal frequency value to use.
- `amplitude` is the amplitude of the vibration between 0.0 and 1.0.

The XrHapticVibration is used in calls to xrApplyHapticFeedback that trigger `vibration` output actions.

The `duration`, and `frequency` parameters **may** be clamped to implementation-dependent ranges.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_HAPTIC_VIBRATION`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

XR_MIN_HAPTIC_DURATION is used to indicate to the runtime that a short haptic pulse of the minimal supported duration for the haptic device.

```
#define XR_MIN_HAPTIC_DURATION -1
```

XR_FREQUENCY_UNSPECIFIED is used to indicate that the application wants the runtime to decide what the optimal frequency is for the haptic pulse.

```
#define XR_FREQUENCY_UNSPECIFIED 0
```

The xrStopHapticFeedback function is defined as:

```
XrResult xrStopHapticFeedback(
    XrSession                                   session,
    const XrHapticActionInfo*                   hapticActionInfo);
```

Parameter Descriptions

- `session` is the XrSession to stop outputting to.
- `hapticActionInfo` is a pointer to an XrHapticActionInfo to provide action and subaction path information.

If a haptic event from this XrAction is in progress, when this function is called the runtime **must** stop that event.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- `hapticActionInfo` **must** be a pointer to a valid XrHapticActionInfo structure

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_ACTIONSET_NOT_ATTACHED
- XR_ERROR_ACTION_TYPE_MISMATCH
- XR_ERROR_PATH_INVALID
- XR_ERROR_PATH_UNSUPPORTED

11.7. Input Action State Synchronization

The xrSyncActions function is defined as:

```
XrResult xrSyncActions(
    XrSession                                   session,
    const XrActionsSyncInfo*                    syncInfo);
```

Parameter Descriptions

- session is a handle to the XrSession that all provided action set handles belong to.
- syncInfo is an XrActionsSyncInfo providing information to synchronize action states.

xrSyncActions updates the current state of input actions. Repeated input action state queries between subsequent synchronization calls **must** return the same values. The XrActionSet structures referenced in the syncInfo.activeActionSets **must** have been previously attached to the session via xrAttachSessionActionSets. If any action sets not attached to this session are passed to xrSyncActions it **must** return XR_ERROR_ACTIONSET_NOT_ATTACHED. Subsets of the bound action sets **can** be synchronized in order to control which actions are seen as active.

If session is not focused, the runtime **must** return XR_SESSION_NOT_FOCUSED, and all action states in the session **must** be inactive.

Valid Usage (Implicit)

- session **must** be a valid XrSession handle
- syncInfo **must** be a pointer to a valid XrActionsSyncInfo structure

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING
- XR_SESSION_NOT_FOCUSED

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_ACTIONSET_NOT_ATTACHED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_PATH_INVALID
- XR_ERROR_PATH_UNSUPPORTED

The XrActionsSyncInfo structure is defined as:

```
typedef struct XrActionsSyncInfo {
    XrStructureType             type;
    const void*                 next;
    uint32_t                    countActiveActionSets;
    const XrActiveActionSet*    activeActionSets;
} XrActionsSyncInfo;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- countActiveActionSets is an integer specifying the number of valid elements in the activeActionSets array.
- activeActionSets is NULL or a pointer to an array of one or more XrActiveActionSet structures that should be synchronized.

Valid Usage (Implicit)

- type **must** be XR_TYPE_ACTIONS_SYNC_INFO
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- If countActiveActionSets is not 0, activeActionSets **must** be a pointer to an array of countActiveActionSets valid XrActiveActionSet structures

The XrActiveActionSet structure is defined as:

```
typedef struct XrActiveActionSet {
    XrActionSet    actionSet;
    XrPath         subactionPath;
} XrActiveActionSet;
```

Member Descriptions

- actionSet is the handle of the action set to activate.
- subactionPath is a subaction path that was declared when one or more actions in the action set was created or XR_NULL_PATH. If the application wants to activate the action set on more than one subaction path, it **can** include additional XrActiveActionSet structs with the other subactionPath values. Using XR_NULL_PATH as the value for subactionPath, acts as a wildcard for all subaction paths on the actions in the action set. If the subaction path was not specified on any of the actions in the actionSet when that action was created, the runtime **must** return XR_ERROR_PATH_UNSUPPORTED.

This structure defines a single active action set and subaction path combination. Applications **can** provide a list of these structures to the xrSyncActions function.

Valid Usage (Implicit)

- `actionSet` **must** be a valid XrActionSet handle

11.8. Action Sources

An application **can** use the xrEnumerateBoundSourcesForAction and xrGetInputSourceLocalizedName calls to prompt the user which physical inputs to use in order to perform an action. A **source** is the physical control that the action is bound to within the current interaction profile as returned by xrGetCurrentInteractionProfile. An action **may** be bound to multiple sources at one time, for example an action named `hold` could be bound to both the X and A buttons.

Once the semantic paths for the action's source are obtained, the application **can** gather additional information about the source. xrGetInputSourceLocalizedName returns a localized human-readable string describing the source, e.g. 'A Button'.

The xrEnumerateBoundSourcesForAction function is defined as:

```
XrResult xrEnumerateBoundSourcesForAction(
    XrSession                                   session,
    const XrBoundSourcesForActionEnumerateInfo* enumerateInfo,
    uint32_t                                    sourceCapacityInput,
    uint32_t*                                   sourceCountOutput,
    XrPath*                                     sources);
```

Parameter Descriptions

- `session` is the XrSession being queried.
- `enumerateInfo` is an XrBoundSourcesForActionEnumerateInfo providing the query information.
- `sourceCapacityInput` is the capacity of the array, or 0 to indicate a request to retrieve the required capacity.
- `sourceCountOutput` is a pointer to the count of sources, or a pointer to the required capacity in the case that `sourceCapacityInput` is 0.
- `sources` is a pointer to an application-allocated array that will be filled with the `XrPath` values for all sources. It **can** be `NULL` if `sourceCapacityInput` is 0.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required `sources` size.

If an action is unbound, xrEnumerateBoundSourcesForAction **must** assign `0` to the value pointed-to by `sourceCountOutput` and not modify the array.

xrEnumerateBoundSourcesForAction **must** return `XR_ERROR_ACTIONSET_NOT_ATTACHED` if passed an action in an action set never attached to the session with xrAttachSessionActionSets.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- `enumerateInfo` **must** be a pointer to a valid XrBoundSourcesForActionEnumerateInfo structure
- `sourceCountOutput` **must** be a pointer to a `uint32_t` value
- If `sourceCapacityInput` is not `0`, `sources` **must** be a pointer to an array of `sourceCapacityInput` `XrPath` values

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_ACTIONSET_NOT_ATTACHED
- XR_ERROR_SIZE_INSUFFICIENT
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_PATH_INVALID

The XrBoundSourcesForActionEnumerateInfo structure is defined as:

```
typedef struct XrBoundSourcesForActionEnumerateInfo {
    XrStructureType    type;
    const void*        next;
    XrAction           action;
} XrBoundSourcesForActionEnumerateInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `action` is the handle of the action to query.

Valid Usage (Implicit)

- type **must** be XR_TYPE_BOUND_SOURCES_FOR_ACTION_ENUMERATE_INFO
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- action **must** be a valid XrAction handle

The xrGetInputSourceLocalizedName function is defined as:

```
XrResult xrGetInputSourceLocalizedName(
    XrSession                                   session,
    const XrInputSourceLocalizedNameGetInfo*    getInfo,
    uint32_t                                    bufferCapacityInput,
    uint32_t*                                   bufferCountOutput,
    char*                                       buffer);
```

Parameter Descriptions

- session is a handle to the XrSession associated with the action that reported this source.
- getInfo is an XrInputSourceLocalizedNameGetInfo providing the query information.
- bufferCapacityInput is the capacity of the buffer, or 0 to indicate a request to retrieve the required capacity.
- bufferCountOutput is a pointer to the count of name characters written (including the terminating \0), or a pointer to the required capacity in the case that bufferCapacityInput is 0.
- buffer is a pointer to an application-allocated buffer that will be filled with the source name. It **can** be NULL if bufferCapacityInput is 0.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required buffer size.

xrGetInputSourceLocalizedName returns a string for the input source in the current system locale.

If xrAttachSessionActionSets has not yet been called for the session, the runtime **must** return XR_ERROR_ACTIONSET_NOT_ATTACHED.

Valid Usage (Implicit)

- `session` **must** be a valid XrSession handle
- `getInfo` **must** be a pointer to a valid XrInputSourceLocalizedNameGetInfo structure
- `bufferCountOutput` **must** be a pointer to a `uint32_t` value
- If `bufferCapacityInput` is not `0`, `buffer` **must** be a pointer to an array of `bufferCapacityInput` char values

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_SIZE_INSUFFICIENT`
- `XR_ERROR_PATH_INVALID`
- `XR_ERROR_PATH_UNSUPPORTED`
- `XR_ERROR_ACTIONSET_NOT_ATTACHED`

The XrInputSourceLocalizedNameGetInfo structure is defined as:

```
typedef struct XrInputSourceLocalizedNameGetInfo {
    XrStructureType                     type;
    const void*                         next;
    XrPath                              sourcePath;
    XrInputSourceLocalizedNameFlags    whichComponents;
} XrInputSourceLocalizedNameGetInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `source` is an `XrPath` representing the source. Typically this was returned by a call to xrEnumerateBoundSourcesForAction.
- `whichComponents` is any set of flags from XrInputSourceLocalizedNameFlagBits.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_INPUT_SOURCE_LOCALIZED_NAME_GET_INFO`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `whichComponents` **must** be a valid combination of XrInputSourceLocalizedNameFlagBits values
- `whichComponents` **must** not be `0`

The xrGetInputSourceLocalizedName::`whichComponents` parameter takes bitwise-OR of any of the following values:

```
// Flag bits for XrInputSourceLocalizedNameFlags
static const XrInputSourceLocalizedNameFlags XR_INPUT_SOURCE_LOCALIZED_NAME_USER_PATH_BIT
= 0x00000001;
static const XrInputSourceLocalizedNameFlags
XR_INPUT_SOURCE_LOCALIZED_NAME_INTERACTION_PROFILE_BIT = 0x00000002;
static const XrInputSourceLocalizedNameFlags XR_INPUT_SOURCE_LOCALIZED_NAME_COMPONENT_BIT
= 0x00000004;
```

Flag Descriptions

- `XR_INPUT_SOURCE_LOCALIZED_NAME_USER_PATH_BIT` indicates that the runtime **must** include the user path portion of the string in the result, if available. E.g. `Left Hand`.
- `XR_INPUT_SOURCE_LOCALIZED_NAME_INTERACTION_PROFILE_BIT` indicates that the runtime **must** include the interaction profile portion of the string in the result, if available. E.g. `Vive Controller`.
- `XR_INPUT_SOURCE_LOCALIZED_NAME_COMPONENT_BIT` indicates that the runtime **must** include the input component portion of the string in the result, if available. E.g. `Trigger`.

Chapter 12. List of Extensions

- XR_KHR_android_create_instance
- XR_KHR_android_surface_swapchain
- XR_KHR_android_thread_settings
- XR_KHR_binding_modification
- XR_KHR_composition_layer_color_scale_bias
- XR_KHR_composition_layer_cube
- XR_KHR_composition_layer_cylinder
- XR_KHR_composition_layer_depth
- XR_KHR_composition_layer_equirect
- XR_KHR_composition_layer_equirect2
- XR_KHR_convert_timespec_time
- XR_KHR_D3D11_enable
- XR_KHR_D3D12_enable
- XR_KHR_loader_init
- XR_KHR_loader_init_android
- XR_KHR_opengl_enable
- XR_KHR_opengl_es_enable
- XR_KHR_visibility_mask
- XR_KHR_vulkan_enable
- XR_KHR_vulkan_enable2
- XR_KHR_vulkan_swapchain_format_list
- XR_KHR_win32_convert_performance_counter_time
- XR_EXT_conformance_automation
- XR_EXT_debug_utils
- XR_EXT_eye_gaze_interaction
- XR_EXT_hand_joints_motion_range
- XR_EXT_hand_tracking
- XR_EXT_hp_mixed_reality_controller
- XR_EXT_performance_settings
- XR_EXT_samsung_odyssey_controller

- XR_EXT_thermal_query
- XR_EXT_view_configuration_depth_range
- XR_EXT_win32_appcontainer_compatible
- XR_EPIC_view_configuration_fov
- XR_FB_android_surface_swapchain_create
- XR_FB_color_space
- XR_FB_display_refresh_rate
- XR_FB_swapchain_update_state
- XR_HTC_vive_cosmos_controller_interaction
- XR_HUAWEI_controller_interaction
- XR_MND_headless
- XR_MND_swapchain_usage_input_attachment_bit
- XR_MSFT_controller_model
- XR_MSFT_first_person_observer
- XR_MSFT_hand_interaction
- XR_MSFT_hand_tracking_mesh
- XR_MSFT_holographic_window_attachment
- XR_MSFT_perception_anchor_interop
- XR_MSFT_secondary_view_configuration
- XR_MSFT_spatial_anchor
- XR_MSFT_spatial_graph_bridge
- XR_MSFT_unbounded_reference_space
- XR_OCULUS_android_session_state_enable
- XR_VALVE_analog_threshold
- XR_VARJO_composition_layer_depth_test
- XR_VARJO_environment_depth_estimation
- XR_VARJO_foveated_rendering
- XR_VARJO_quad_views <<<

12.1. XR_KHR_android_create_instance

Name String

`XR_KHR_android_create_instance`

Extension Type

Instance extension

Registered Extension Number

9

Revision

3

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2019-07-17

IP Status

No known IP claims.

Contributors

Robert Menzel, NVIDIA
Martin Renschler, Qualcomm
Krzysztof Kosiński, Google

Overview

When the application creates an XrInstance object on Android systems, additional information from the application has to be provided to the XR runtime.

The Android XR runtime **must** return error `XR_ERROR_VALIDATION_FAILURE` if the additional information is not provided by the application or if the additional parameters are invalid.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_INSTANCE_CREATE_INFO_ANDROID_KHR`

New Enums

New Structures

The XrInstanceCreateInfoAndroidKHR structure is defined as:

```
typedef struct XrInstanceCreateInfoAndroidKHR {
    XrStructureType    type;
    const void*        next;
    void*              applicationVM;
    void*              applicationActivity;
} XrInstanceCreateInfoAndroidKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `applicationVM` is a pointer to the JNI's opaque `JavaVM` structure, cast to a void pointer.
- `applicationActivity` is a JNI reference to an `android.app.Activity` that will drive the session lifecycle of this instance, cast to a void pointer.

XrInstanceCreateInfoAndroidKHR contains additional Android specific information needed when calling xrCreateInstance. The `applicationVM` field should be populated with the `JavaVM` structure received by the `JNI_OnLoad` function, while the `applicationActivity` field will typically contain a reference to a Java activity object received through an application-specific native method. The XrInstanceCreateInfoAndroidKHR structure **must** be provided in the `next` chain of the XrInstanceCreateInfo structure when calling xrCreateInstance.

Valid Usage (Implicit)

- The `XR_KHR_android_create_instance` extension **must** be enabled prior to using XrInstanceCreateInfoAndroidKHR
- `type` **must** be `XR_TYPE_INSTANCE_CREATE_INFO_ANDROID_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `applicationVM` **must** be a pointer value
- `applicationActivity` **must** be a pointer value

New Functions

Issues

Version History

* Revision 1, 2017-05-26 (Robert Menzel)
  * Initial draft
* Revision 2, 2019-01-24 (Martin Renschler)
  * Added error code, reformatted
* Revision 3, 2019-07-17 (Krzysztof Kosiński)
  * Non-substantive clarifications.

12.2. XR_KHR_android_surface_swapchain

Name String

`XR_KHR_android_surface_swapchain`

Extension Type

Instance extension

Registered Extension Number

5

Revision

4

Extension and Version Dependencies

* Requires OpenXR 1.0

Last Modified Date

2019-05-30

IP Status

No known IP claims.

Contributors

Krzysztof Kosiński, Google
Johannes van Waveren, Oculus
Martin Renschler, Qualcomm

Overview

A common activity in XR is to view an image stream. Image streams are often the result of camera previews or decoded video streams. On Android, the basic primitive representing the producer end of an image queue is the class `android.view.Surface`. This extension provides a special swapchain that uses an `android.view.Surface` as its producer end.

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

New Functions

To create an XrSwapchain object and an Android Surface object call:

```
XrResult xrCreateSwapchainAndroidSurfaceKHR(
    XrSession                                   session,
    const XrSwapchainCreateInfo*                info,
    XrSwapchain*                                swapchain,
    jobject*                                    surface);
```

Parameter Descriptions

- `session` is an XrSession handle previously created with xrCreateSession.
- `info` is a pointer to an XrSwapchainCreateInfo structure.
- `swapchain` is a pointer to a handle in which the created XrSwapchain is returned.
- `surface` is a pointer to a `jobject` where the created Android Surface is returned.

xrCreateSwapchainAndroidSurfaceKHR creates an XrSwapchain object returned in `swapchain` and an Android Surface `jobject` returned in `surface`. The `jobject` **must** be valid to be passed back to Java code using JNI and **must** be valid to be used with ordinary Android APIs for submitting images to Surfaces. The returned XrSwapchain **must** be valid to be referenced in XrSwapchainSubImage structures to show content on the screen. The width and height passed in XrSwapchainCreateInfo **may** not be persistent throughout the life cycle of the created swapchain, since on Android, the size of the images is controlled by the producer and possibly changes at any time.

The only function that is allowed to be called on the XrSwapchain returned from this function is xrDestroySwapchain. For example, calling any of the functions xrEnumerateSwapchainImages, xrAcquireSwapchainImage, xrWaitSwapchainImage or xrReleaseSwapchainImage is invalid.

When the application receives the XrEventDataSessionStateChanged event with the `XR_SESSION_STATE_STOPPING` state, it **must** ensure that no threads are writing to any of the Android surfaces created with this extension before calling xrEndSession. The effect of writing frames to the Surface when the session is in states other than `XR_SESSION_STATE_VISIBLE` or `XR_SESSION_STATE_FOCUSED` is undefined.

xrCreateSwapchainAndroidSurfaceKHR **must** return the same set of error codes as xrCreateSwapchain under the same circumstances, plus XR_ERROR_FUNCTION_UNSUPPORTED in case the function is not supported.

Valid Usage of XrSwapchainCreateInfo members

- The XrSwapchainCreateInfo::format,
  XrSwapchainCreateInfo::sampleCount,
  XrSwapchainCreateInfo::faceCount,
  XrSwapchainCreateInfo::arraySize and
  XrSwapchainCreateInfo::mipCount
  members of the structure passed as the info parameter **must** be zero.

Valid Usage (Implicit)

- The XR_KHR_android_surface_swapchain extension **must** be enabled prior to calling xrCreateSwapchainAndroidSurfaceKHR
- session **must** be a valid XrSession handle
- info **must** be a pointer to a valid XrSwapchainCreateInfo structure
- swapchain **must** be a pointer to an XrSwapchain handle
- surface **must** be a pointer to a jobject value

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_LIMIT_REACHED
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_FUNCTION_UNSUPPORTED

Issues

Version History

- Revision 1, 2017-01-17 (Johannes van Waveren)
  - Initial draft
- Revision 2, 2017-10-30 (Kaye Mason)
  - Changed images to swapchains, used snippet includes. Added issue for Surfaces.
- Revision 3, 2018-05-16 (Krzysztof Kosiński)
  - Refactored to use Surface instead of SurfaceTexture.
- Revision 4, 2019-01-24 (Martin Renschler)
  - Refined the specification of the extension

12.3. XR_KHR_android_thread_settings

Name String

`XR_KHR_android_thread_settings`

Extension Type

Instance extension

Registered Extension Number

4

Revision

5

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2019-07-17

IP Status

No known IP claims.

Contributors

Cass Everitt, Oculus
Johannes van Waveren, Oculus
Martin Renschler, Qualcomm
Krzysztof Kosiński, Google

Overview

For XR to be comfortable, it is important for applications to deliver frames quickly and consistently. In order to make sure the important application threads get their full share of time, these threads must be identified to the system, which will adjust their scheduling priority accordingly.

New Object Types

New Flag Types

New Enum Constants

XrResult enumeration is extended with:

- XR_ERROR_ANDROID_THREAD_SETTINGS_ID_INVALID_KHR
- XR_ERROR_ANDROID_THREAD_SETTINGS_FAILURE_KHR

New Enums

The possible thread types are specified by the XrAndroidThreadTypeKHR enumeration:

```
typedef enum XrAndroidThreadTypeKHR {
    XR_ANDROID_THREAD_TYPE_APPLICATION_MAIN_KHR = 1,
    XR_ANDROID_THREAD_TYPE_APPLICATION_WORKER_KHR = 2,
    XR_ANDROID_THREAD_TYPE_RENDERER_MAIN_KHR = 3,
    XR_ANDROID_THREAD_TYPE_RENDERER_WORKER_KHR = 4,
    XR_ANDROID_THREAD_TYPE_MAX_ENUM_KHR = 0x7FFFFFFF
} XrAndroidThreadTypeKHR;
```

Enumerants

- XR_ANDROID_THREAD_TYPE_APPLICATION_MAIN_KHR
  hints the XR runtime that the thread is doing background CPU tasks
- XR_ANDROID_THREAD_TYPE_APPLICATION_WORKER_KHR
  hints the XR runtime that the thread is doing time critical CPU tasks
- XR_ANDROID_THREAD_TYPE_RENDERER_MAIN_KHR
  hints the XR runtime that the thread is doing background graphics device tasks
- XR_ANDROID_THREAD_TYPE_RENDERER_WORKER_KHR
  hints the XR runtime that the thread is doing time critical graphics device tasks

New Structures

New Functions

To declare a thread to be of a certain XrAndroidThreadTypeKHR type call:

```
XrResult xrSetAndroidApplicationThreadKHR(
    XrSession                                   session,
    XrAndroidThreadTypeKHR                      threadType,
    uint32_t                                    threadId);
```

Parameter Descriptions

- `session` is a valid XrSession handle.
- `threadType` is a classification of the declared thread allowing the XR runtime to apply the relevant priority and attributes. If such settings fail, the runtime **must** return `XR_ERROR_ANDROID_THREAD_SETTINGS_FAILURE_KHR`.
- `threadId` is the kernel thread ID of the declared thread, as returned by `gettid()` or `android.os.process.myTid()`. If the thread ID is invalid, the runtime **must** return `XR_ERROR_ANDROID_THREAD_SETTINGS_ID_INVALID_KHR`.

xrSetAndroidApplicationThreadKHR allows to declare an XR-critical thread and to classify it.

Valid Usage (Implicit)

- The `XR_KHR_android_thread_settings` extension **must** be enabled prior to calling xrSetAndroidApplicationThreadKHR
- `session` **must** be a valid XrSession handle
- `threadType` **must** be a valid XrAndroidThreadTypeKHR value

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_ANDROID_THREAD_SETTINGS_ID_INVALID_KHR
- XR_ERROR_ANDROID_THREAD_SETTINGS_FAILURE_KHR
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_FUNCTION_UNSUPPORTED

Version History

- Revision 1, 2017-01-17 (Johannes van Waveren)
  - Initial draft.
- Revision 2, 2017-10-31 (Armelle Laine)
  - Move the performance settings to EXT extension.
- Revision 3, 2018-12-20 (Paul Pedriana)
  - Revised the error code naming to use KHR and renamed xrSetApplicationThreadKHR → xrSetAndroidApplicationThreadKHR.
- Revision 4, 2019-01-24 (Martin Renschler)
  - Added enum specification, reformatting
- Revision 5, 2019-07-17 (Krzysztof Kosiński)
  - Clarify the type of thread identifier used by the extension.

12.4. XR_KHR_binding_modification

Name String

XR_KHR_binding_modification

Extension Type

Instance extension

Registered Extension Number

121

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2020-07-29

IP Status

No known IP claims.

Contributors

Joe Ludwig, Valve

Contacts

Joe Ludwig, Valve

Overview

This extension adds an optional structure that can be included on the XrInteractionProfileSuggestedBinding::`next` chain passed to xrSuggestInteractionProfileBindings to specify additional information to modify default binding behavior.

This extension does not define any actual modification structs, but includes the list of modifications and the XrBindingModificationBaseHeaderKHR structure to allow other extensions to provide specific modifications.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_BINDING_MODIFICATIONS_KHR`

New Enums

New Structures

The XrBindingModificationsKHR structure is defined as:

```
typedef struct XrBindingModificationsKHR {
    XrStructureType                                type;
    const void*                                    next;
    uint32_t                                       bindingModificationCount;
    const XrBindingModificationBaseHeaderKHR* const*    bindingModifications;
} XrBindingModificationsKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain.
- `bindingModificationCount` is the number of binding modifications in the array pointed to by `bindingModifications`.
- `bindingModifications` is a pointer to an array of pointers to binding modification structures based on XrBindingModificationBaseHeaderKHR, that define all of the application's suggested binding modifications for the specified interaction profile.

Valid Usage (Implicit)

- The `XR_KHR_binding_modification` extension **must** be enabled prior to using XrBindingModificationsKHR
- `type` **must** be `XR_TYPE_BINDING_MODIFICATIONS_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- If `bindingModificationCount` is not `0`, `bindingModifications` **must** be a pointer to an array of `bindingModificationCount` valid XrBindingModificationBaseHeaderKHR-based structures

The XrBindingModificationBaseHeaderKHR structure is defined as:

```
typedef struct XrBindingModificationBaseHeaderKHR {
    XrStructureType    type;
    const void*        next;
} XrBindingModificationBaseHeaderKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure. This base structure itself has no associated XrStructureType value.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or in this extension.

The XrBindingModificationBaseHeaderKHR is a base structure is overridden by `XrBindingModification*` child structures.

Valid Usage (Implicit)

- The `XR_KHR_binding_modification` extension **must** be enabled prior to using XrBindingModificationBaseHeaderKHR
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

New Functions

Issues

Version History

- Revision 1, 2020-08-06 (Joe Ludwig)
  - Initial draft.

12.5. XR_KHR_composition_layer_color_scale_bias

Name String

`XR_KHR_composition_layer_color_scale_bias`

Extension Type

Instance extension

Registered Extension Number

35

Revision

5

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2019-01-28

IP Status

No known IP claims.

Contributors

Paul Pedriana, Oculus
Cass Everitt, Oculus
Martin Renschler, Qualcomm

Overview

Color scale and bias are applied to a layer color during composition, after its conversion to premultiplied alpha representation.

If specified, `colorScale` and `colorBias` **must** be used to alter the LayerColor as follows:

- colorScale = max(vec4(0, 0, 0, 0), colorScale)
- LayerColor.RGB = LayerColor.A > 0 ? LayerColor.RGB / LayerColor.A : vec3(0, 0, 0)
- LayerColor = LayerColor * colorScale + colorBias
- LayerColor.RGB *= LayerColor.A

This extension specifies the XrCompositionLayerColorScaleBiasKHR structure, which, if present in the XrCompositionLayerBaseHeader::`next` chain, **must** be applied to the composition layer.

This extension does not define a new composition layer type, but rather it defines a transform that may be applied to the color derived from existing composition layer types.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_COMPOSITION_LAYER_COLOR_SCALE_BIAS_KHR`

New Enums

New Structures

The XrCompositionLayerColorScaleBiasKHR structure is defined as:

```
typedef struct XrCompositionLayerColorScaleBiasKHR {
    XrStructureType    type;
    const void*        next;
    XrColor4f          colorScale;
    XrColor4f          colorBias;
} XrCompositionLayerColorScaleBiasKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `colorScale` is an XrColor4f which will modulate the color sourced from the images.
- `colorBias` is an XrColor4f which will offset the color sourced from the images.

XrCompositionLayerColorScaleBiasKHR contains the information needed to scale and bias the color of layer textures.

The XrCompositionLayerColorScaleBiasKHR structure **can** be applied by applications to composition layers by adding an instance of the struct to the XrCompositionLayerBaseHeader::`next` list.

Valid Usage (Implicit)

- The `XR_KHR_composition_layer_color_scale_bias` extension **must** be enabled prior to using XrCompositionLayerColorScaleBiasKHR
- `type` **must** be `XR_TYPE_COMPOSITION_LAYER_COLOR_SCALE_BIAS_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

New Functions

Issues

Version History

- Revision 1, 2017-09-13 (Paul Pedriana)
  - Initial implementation.
- Revision 2, 2019-01-24 (Martin Renschler)
  - Formatting, spec language changes
- Revision 3, 2019-01-28 (Paul Pedriana)

    ◦ Revised math to remove premultiplied alpha before applying color scale and offset, then restoring.

* Revision 4, 2019-07-17 (Cass Everitt)
    ◦ Non-substantive updates to the spec language and equations.
* Revision 5, 2020-05-20 (Cass Everitt)
    ◦ Changed extension name, simplified language.

12.6. XR_KHR_composition_layer_cube

Name String

`XR_KHR_composition_layer_cube`

Extension Type

Instance extension

Registered Extension Number

7

Revision

8

Extension and Version Dependencies

* Requires OpenXR 1.0

Last Modified Date

2019-01-24

IP Status

No known IP claims.

Contributors

Johannes van Waveren, Oculus
Cass Everitt, Oculus
Paul Pedriana, Oculus
Gloria Kennickell, Oculus
Sam Martin, ARM
Kaye Mason, Google, Inc.
Martin Renschler, Qualcomm

Contacts

Cass Everitt, Oculus
Paul Pedriana, Oculus

Overview

This extension adds an additional layer type that enables direct sampling from cubemaps.

The cube layer is the natural layer type for hardware accelerated environment maps. Without updating the image source, the user can look all around, and the compositor can display what they are looking at without intervention from the application.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_COMPOSITION_LAYER_CUBE_KHR`

New Enums

New Structures

The XrCompositionLayerCubeKHR structure is defined as:

```
typedef struct XrCompositionLayerCubeKHR {
    XrStructureType             type;
    const void*                 next;
    XrCompositionLayerFlags     layerFlags;
    XrSpace                     space;
    XrEyeVisibility             eyeVisibility;
    XrSwapchain                 swapchain;
    uint32_t                    imageArrayIndex;
    XrQuaternionf               orientation;
} XrCompositionLayerCubeKHR;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- layerFlags is any flags to apply to this layer.
- space is the XrSpace in which the orientation of the cube layer is evaluated over time.
- eye is the eye represented by this layer.
- swapchain is the swapchain.
- imageArrayIndex is the image array index, with 0 meaning the first or only array element.
- orientation is the orientation of the environment map in the space.

XrCompositionLayerCubeKHR contains the information needed to render a cube map when calling xrEndFrame. XrCompositionLayerCubeKHR is an alias type for the base struct XrCompositionLayerBaseHeader used in XrFrameEndInfo.

Valid Usage (Implicit)

- The XR_KHR_composition_layer_cube extension **must** be enabled prior to using XrCompositionLayerCubeKHR
- type **must** be XR_TYPE_COMPOSITION_LAYER_CUBE_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- layerFlags **must** be 0 or a valid combination of XrCompositionLayerFlagBits values
- space **must** be a valid XrSpace handle
- eyeVisibility **must** be a valid XrEyeVisibility value
- swapchain **must** be a valid XrSwapchain handle
- Both of space and swapchain **must** have been created, allocated, or retrieved from the same XrSession

New Functions

Issues

Version History

- Revision 0, 2017-02-01 (Johannes van Waveren)
  - Initial draft.

- Revision 1, 2017-05-19 (Sam Martin)
  - Initial draft, moving the 3 layer types to an extension.
- Revision 2, 2017-08-30 (Paul Pedriana)
  - Updated the specification.
- Revision 3, 2017-10-12 (Cass Everitt)
  - Updated to reflect per-eye structs and the change to swapchains
- Revision 4, 2017-10-18 (Kaye Mason)
  - Update to flatten structs to remove per-eye arrays.
- Revision 5, 2017-12-05 (Paul Pedriana)
  - Updated to break out the cylinder and equirect features into separate extensions.
- Revision 6, 2017-12-07 (Paul Pedriana)
  - Updated to use transform components instead of transform matrices.
- Revision 7, 2017-12-07 (Paul Pedriana)
  - Updated to convert XrPosef to XrQuaternionf (there's no position component).
- Revision 8, 2019-01-24 (Martin Renschler)
  - Updated struct to use XrSwapchainSubImage, reformat and spec language changes, eye parameter description update

12.7. XR_KHR_composition_layer_cylinder

Name String

`XR_KHR_composition_layer_cylinder`

Extension Type

Instance extension

Registered Extension Number

18

Revision

4

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2019-01-24

IP Status

No known IP claims.

Contributors

James Hughes, Oculus
Paul Pedriana, Oculus
Martin Renschler, Qualcomm

Contacts

Paul Pedriana, Oculus
Cass Everitt, Oculus

Overview

This extension adds an additional layer type where the XR runtime **must** map a texture stemming from a swapchain onto the inside of a cylinder section. It can be imagined much the same way a curved television display looks to a viewer. This is not a projection type of layer but rather an object-in-world type of layer, similar to XrCompositionLayerQuad. Only the interior of the cylinder surface **must** be visible; the exterior of the cylinder is not visible and **must** not be drawn by the runtime.

The cylinder characteristics are specified by the following parameters:

```
XrPosef          pose;
float            radius;
float            centralAngle;
float            aspectRatio;
```

These can be understood via the following diagram, which is a top-down view of a horizontally oriented cylinder. The aspect ratio drives how tall the cylinder will appear based on the other parameters. Typically the aspectRatio would be set to be the aspect ratio of the texture being used, so that it looks the same within the cylinder as it does in 2D.

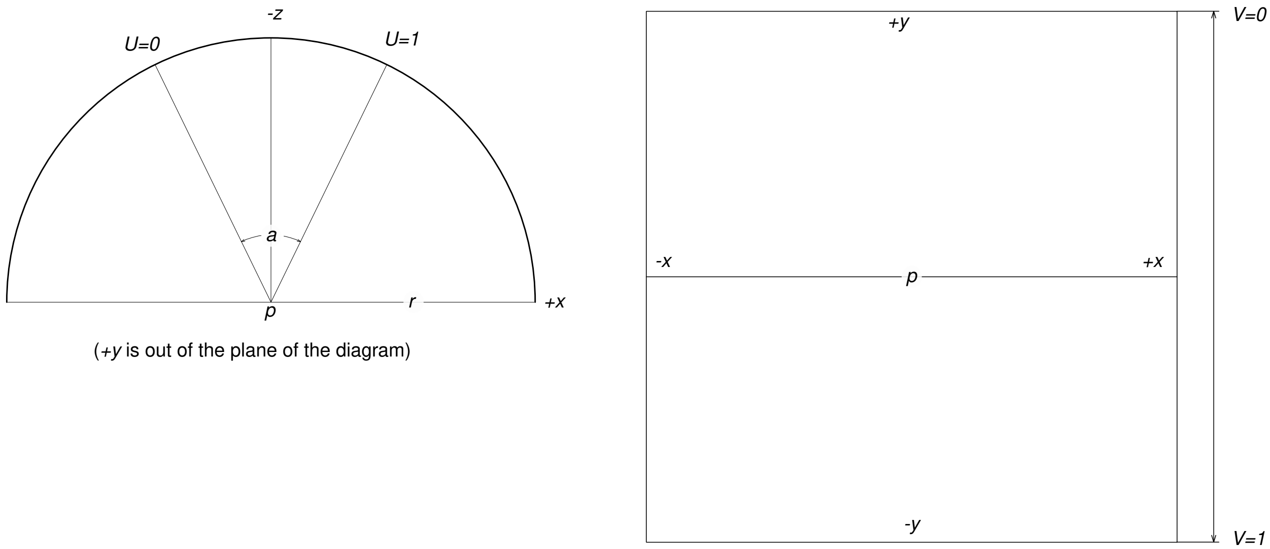


Figure 3. Cylinder Layer Parameters

- r — Radius
- a — Central angle in (0, 2π)
- p — Origin of pose transform
- U/V — UV coordinates

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_COMPOSITION_LAYER_CYLINDER_KHR`

New Enums

New Structures

The XrCompositionLayerCylinderKHR structure is defined as:

```
typedef struct XrCompositionLayerCylinderKHR {
    XrStructureType            type;
    const void*                next;
    XrCompositionLayerFlags    layerFlags;
    XrSpace                    space;
    XrEyeVisibility            eyeVisibility;
    XrSwapchainSubImage        subImage;
    XrPosef                    pose;
    float                      radius;
    float                      centralAngle;
    float                      aspectRatio;
} XrCompositionLayerCylinderKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `layerFlags` specifies options for the layer.
- `space` is the XrSpace in which the `pose` of the cylinder layer is evaluated over time.
- `eye` is the eye represented by this layer.
- `subImage` identifies the image XrSwapchainSubImage to use.
- `pose` is an XrPosef defining the position and orientation of the center point of the view of the cylinder within the reference frame of the `space`.
- `radius` is the non-negative radius of the cylinder. Values of zero or floating point positive infinity are treated as an infinite cylinder.
- `centralAngle` is the angle of the visible section of the cylinder, based at 0 radians, in the range of $[0, 2\pi)$. It grows symmetrically around the 0 radian angle.
- `aspectRatio` is the ratio of the visible cylinder section width / height. The height of the cylinder is given by: (cylinder radius × cylinder angle) / aspectRatio.

XrCompositionLayerCylinderKHR contains the information needed to render a texture onto a cylinder when calling xrEndFrame. XrCompositionLayerCylinderKHR is an alias type for the base struct XrCompositionLayerBaseHeader used in XrFrameEndInfo.

Valid Usage (Implicit)

- The `XR_KHR_composition_layer_cylinder` extension **must** be enabled prior to using XrCompositionLayerCylinderKHR
- `type` **must** be `XR_TYPE_COMPOSITION_LAYER_CYLINDER_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `layerFlags` **must** be `0` or a valid combination of XrCompositionLayerFlagBits values
- `space` **must** be a valid XrSpace handle
- `eyeVisibility` **must** be a valid XrEyeVisibility value
- `subImage` **must** be a valid XrSwapchainSubImage structure

New Functions

Issues

Version History

- Revision 1, 2017-05-19 (Paul Pedriana)
    - Initial version. This was originally part of a single extension which supported multiple such extension layer types.
- Revision 2, 2017-12-07 (Paul Pedriana)
    - Updated to use transform components instead of transform matrices.
- Revision 3, 2018-03-05 (Paul Pedriana)
    - Added improved documentation and brought the documentation in line with the existing core spec.
- Revision 4, 2019-01-24 (Martin Renschler)
    - Reformatted, spec language changes, eye parameter description update

12.8. XR_KHR_composition_layer_depth

Name String

`XR_KHR_composition_layer_depth`

Extension Type

Instance extension

Registered Extension Number

11

Revision

5

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2019-01-24

IP Status

No known IP claims.

Contributors

Paul Pedriana, Oculus
Bryce Hutchings, Microsoft
Andreas Loeve Selvik, Arm
Martin Renschler, Qualcomm

Overview

This extension defines an extra layer type which allows applications to submit valid depth buffers along with images submitted in projection layers, i.e. XrCompositionLayerProjection.

The XR runtime **may** use this information to perform more accurate reprojections taking depth into account. Use of this extension does not affect the order of layer composition as described in Compositing.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_COMPOSITION_LAYER_DEPTH_INFO_KHR`

New Enums

New Structures

When submitting depth buffers along with projection layers, add the XrCompositionLayerDepthInfoKHR to the `next` chain for all XrCompositionLayerProjectionView structures in the given layer.

The XrCompositionLayerDepthInfoKHR structure is defined as:

```
typedef struct XrCompositionLayerDepthInfoKHR {
    XrStructureType       type;
    const void*           next;
    XrSwapchainSubImage   subImage;
    float                 minDepth;
    float                 maxDepth;
    float                 nearZ;
    float                 farZ;
} XrCompositionLayerDepthInfoKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `subImage` identifies the depth image XrSwapchainSubImage to be associated with the color swapchain. The contained `imageRect` specifies the valid portion of the depth image to use, in pixels. It also implicitly defines the transform from normalized image coordinates into pixel coordinates. The contained `imageArrayIndex` is the depth image array index, with 0 meaning the first or only array element.
- `minDepth` and `maxDepth` are the range of depth values the `depthSwapchain` could have, in the range of [0.0,1.0]. This is akin to min and max values of OpenGL's `glDepthRange`, but with the requirement here that maxDepth ≥ minDepth.
- `nearZ` is the positive distance in meters of the `minDepth` value in the depth swapchain. Applications **may** use a `nearZ` that is greater than `farZ` to indicate depth values are reversed. `nearZ` can be infinite.
- `farZ` is the positive distance in meters of the `maxDepth` value in the depth swapchain. `farZ` can be infinite. Applications **must** not use the same value as `nearZ`.

XrCompositionLayerDepthInfoKHR contains the information needed to specify an extra layer with depth information. When submitting depth buffers along with projection layers, add the XrCompositionLayerDepthInfoKHR to the `next` chain for all XrCompositionLayerProjectionView structures in the given layer.

Valid Usage (Implicit)

- The XR_KHR_composition_layer_depth extension **must** be enabled prior to using XrCompositionLayerDepthInfoKHR
- type **must** be XR_TYPE_COMPOSITION_LAYER_DEPTH_INFO_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- subImage **must** be a valid XrSwapchainSubImage structure

New Functions

Issues

Version History

- Revision 1, 2017-08-18 (Paul Pedriana)
  - Initial proposal.
- Revision 2, 2017-10-30 (Kaye Mason)
  - Migration from Images to Swapchains.
- Revision 3, 2018-07-20 (Bryce Hutchings)
  - Support for swapchain texture arrays
- Revision 4, 2018-12-17 (Andreas Loeve Selvik)
  - depthImageRect in pixels instead of UVs
- Revision 5, 2019-01-24 (Martin Renschler)
  - changed depthSwapchain/depthImageRect/depthImageArrayIndex to XrSwapchainSubImage
  - reformat and spec language changes
  - removed vendor specific terminology

12.9. XR_KHR_composition_layer_equirect

Name String

XR_KHR_composition_layer_equirect

Extension Type

Instance extension

Registered Extension Number

19

Revision

3

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2019-01-24

IP Status

No known IP claims.

Contributors

Johannes van Waveren, Oculus
Cass Everitt, Oculus
Paul Pedriana, Oculus
Gloria Kennickell, Oculus
Martin Renschler, Qualcomm

Contacts

Cass Everitt, Oculus
Paul Pedriana, Oculus

Overview

This extension adds an additional layer type where the XR runtime must map an equirectangular coded image stemming from a swapchain onto the inside of a sphere.

The equirect layer type provides most of the same benefits as a cubemap, but from an equirect 2D image source. This image source is appealing mostly because equirect environment maps are very common, and the highest quality you can get from them is by sampling them directly in the compositor.

This is not a projection type of layer but rather an object-in-world type of layer, similar to XrCompositionLayerQuad. Only the interior of the sphere surface **must** be visible; the exterior of the sphere is not visible and **must** not be drawn by the runtime.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_COMPOSITION_LAYER_EQUIRECT_KHR`

New Enums

New Structures

The XrCompositionLayerEquirectKHR structure is defined as:

```
typedef struct XrCompositionLayerEquirectKHR {
    XrStructureType             type;
    const void*                 next;
    XrCompositionLayerFlags     layerFlags;
    XrSpace                     space;
    XrEyeVisibility             eyeVisibility;
    XrSwapchainSubImage         subImage;
    XrPosef                     pose;
    float                       radius;
    XrVector2f                  scale;
    XrVector2f                  bias;
} XrCompositionLayerEquirectKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `layerFlags` specifies options for the layer.
- `space` is the XrSpace in which the `pose` of the equirect layer is evaluated over time.
- `eye` is the eye represented by this layer.
- `subImage` identifies the image XrSwapchainSubImage to use.
- `pose` is an XrPosef defining the position and orientation of the center point of the sphere onto which the equirect image data is mapped, relative to the reference frame of the `space`.
- `radius` is the non-negative radius of the sphere onto which the equirect image data is mapped. Values of zero or floating point positive infinity are treated as an infinite sphere.
- `scale` is an XrVector2f indicating a scale of the texture coordinates after the mapping to 2D.
- `bias` is an XrVector2f indicating a bias of the texture coordinates after the mapping to 2D.

XrCompositionLayerEquirectKHR contains the information needed to render an equirectangular image onto a sphere when calling xrEndFrame. XrCompositionLayerEquirectKHR is an alias type for the base struct XrCompositionLayerBaseHeader used in XrFrameEndInfo.

Valid Usage (Implicit)

- The XR_KHR_composition_layer_equirect extension **must** be enabled prior to using XrCompositionLayerEquirectKHR
- type **must** be XR_TYPE_COMPOSITION_LAYER_EQUIRECT_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- layerFlags **must** be 0 or a valid combination of XrCompositionLayerFlagBits values
- space **must** be a valid XrSpace handle
- eyeVisibility **must** be a valid XrEyeVisibility value
- subImage **must** be a valid XrSwapchainSubImage structure

New Functions

Issues

Version History

- Revision 1, 2017-05-19 (Paul Pedriana)
  - Initial version. This was originally part of a single extension which supported multiple such extension layer types.
- Revision 2, 2017-12-07 (Paul Pedriana)
  - Updated to use transform components instead of transform matrices.
- Revision 3, 2019-01-24 (Martin Renschler)
  - Reformatted, spec language changes, eye parameter description update

12.10. XR_KHR_composition_layer_equirect2

Name String

XR_KHR_composition_layer_equirect2

Extension Type

Instance extension

Registered Extension Number

92

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2019-01-24

IP Status

No known IP claims.

Contributors

Johannes van Waveren, Oculus
Cass Everitt, Oculus
Paul Pedriana, Oculus
Gloria Kennickell, Oculus
Martin Renschler, Qualcomm

Contacts

Cass Everitt, Oculus

Overview

This extension adds an additional layer type where the XR runtime must map an equirectangular coded image stemming from a swapchain onto the inside of a sphere.

The equirect layer type provides most of the same benefits as a cubemap, but from an equirect 2D image source. This image source is appealing mostly because equirect environment maps are very common, and the highest quality you can get from them is by sampling them directly in the compositor.

This is not a projection type of layer but rather an object-in-world type of layer, similar to XrCompositionLayerQuad. Only the interior of the sphere surface **must** be visible; the exterior of the sphere is not visible and **must** not be drawn by the runtime.

This extension uses a different parameterization more in keeping with the formulation of KHR_composition_layer_cylinder but is functionally equivalent to KHR_composition_layer_equirect.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_COMPOSITION_LAYER_EQUIRECT2_KHR`

New Enums

New Structures

The XrCompositionLayerEquirect2KHR structure is defined as:

```
typedef struct XrCompositionLayerEquirect2KHR {
    XrStructureType             type;
    const void*                 next;
    XrCompositionLayerFlags     layerFlags;
    XrSpace                     space;
    XrEyeVisibility             eyeVisibility;
    XrSwapchainSubImage         subImage;
    XrPosef                     pose;
    float                       radius;
    float                       centralHorizontalAngle;
    float                       upperVerticalAngle;
    float                       lowerVerticalAngle;
} XrCompositionLayerEquirect2KHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `layerFlags` specifies options for the layer.
- `space` is the XrSpace in which the `pose` of the equirect layer is evaluated over time.
- `eye` is the eye represented by this layer.
- `subImage` identifies the image XrSwapchainSubImage to use.
- `pose` is an XrPosef defining the position and orientation of the center point of the sphere onto which the equirect image data is mapped, relative to the reference frame of the `space`.
- `radius` is the non-negative radius of the sphere onto which the equirect image data is mapped. Values of zero or floating point positive infinity are treated as an infinite sphere.
- `centralHorizontalAngle` defines the visible horizontal angle of the sphere, based at 0 radians, in the range of [0, 2π]. It grows symmetrically around the 0 radian angle.
- `upperVerticalAngle` defines the upper vertical angle of the visible portion of the sphere, in the range of [-π/2, π/2].
- `lowerVerticalAngle` defines the lower vertical angle of the visible portion of the sphere, in the range of [-π/2, π/2].

XrCompositionLayerEquirect2KHR contains the information needed to render an equirectangular image onto a sphere when calling xrEndFrame. XrCompositionLayerEquirect2KHR is an alias type for the base struct XrCompositionLayerBaseHeader used in XrFrameEndInfo.

Valid Usage (Implicit)

- The `XR_KHR_composition_layer_equirect2` extension **must** be enabled prior to using XrCompositionLayerEquirect2KHR
- `type` **must** be `XR_TYPE_COMPOSITION_LAYER_EQUIRECT2_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `layerFlags` **must** be `0` or a valid combination of XrCompositionLayerFlagBits values
- `space` **must** be a valid XrSpace handle
- `eyeVisibility` **must** be a valid XrEyeVisibility value
- `subImage` **must** be a valid XrSwapchainSubImage structure

New Functions

Issues

Version History

- Revision 1, 2020-05-08 (Cass Everitt)
  - Initial version.
  - Kept contributors from the original equirect extension.

12.11. XR_KHR_convert_timespec_time

Name String

`XR_KHR_convert_timespec_time`

Extension Type

Instance extension

Registered Extension Number

37

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2019-01-24

IP Status

No known IP claims.

Contributors

Paul Pedriana, Oculus

Overview

This extension provides two functions for converting between timespec monotonic time and `XrTime`. The xrConvertTimespecTimeToTimeKHR function converts from timespec time to `XrTime`, while the xrConvertTimeToTimespecTimeKHR function converts `XrTime` to timespec monotonic time. The primary use case for this functionality is to be able to synchronize events between the local system and the OpenXR system.

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

New Functions

To convert from timespec monotonic time to `XrTime`, call:

```
XrResult xrConvertTimespecTimeToTimeKHR(
    XrInstance                                  instance,
    const struct timespec*                      timespecTime,
    XrTime*                                     time);
```

Parameter Descriptions

- `instance` is an XrInstance handle previously created with xrCreateInstance.
- `unixTime` is a `timespec` obtained from `clock_gettime` with `CLOCK_MONOTONIC`.
- `time` is the resulting `XrTime` that is equivalent to the `unixTime`.

The xrConvertTimespecTimeToTimeKHR function converts a time obtained by the clock_gettime function to the equivalent XrTime.

If the output time cannot represent the input unixTime, the runtime **must** return XR_ERROR_TIME_INVALID.

Valid Usage (Implicit)

- The XR_KHR_convert_timespec_time extension **must** be enabled prior to calling xrConvertTimespecTimeToTimeKHR
- instance **must** be a valid XrInstance handle
- timespecTime **must** be a pointer to a valid timespec value
- time **must** be a pointer to an XrTime value

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_TIME_INVALID
- XR_ERROR_FUNCTION_UNSUPPORTED

To convert from XrTime to timespec monotonic time, call:

```
XrResult xrConvertTimeToTimespecTimeKHR(
    XrInstance                                  instance,
    XrTime                                      time,
    struct timespec*                            timespecTime);
```

Parameter Descriptions

- instance is an XrInstance handle previously created with xrCreateInstance.
- time is an XrTime.
- unixTime is the resulting timespec time that is equivalent to a timespec obtained from clock_gettime with CLOCK_MONOTONIC.

The xrConvertTimeToTimespecTimeKHR function converts an XrTime to time as if generated by clock_gettime.

If the output unixTime cannot represent the input time, the runtime **must** return XR_ERROR_TIME_INVALID.

Valid Usage (Implicit)

- The XR_KHR_convert_timespec_time extension **must** be enabled prior to calling xrConvertTimeToTimespecTimeKHR
- instance **must** be a valid XrInstance handle
- timespecTime **must** be a pointer to a timespec value

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_TIME_INVALID
- XR_ERROR_FUNCTION_UNSUPPORTED

Issues

Version History

- Revision 1, 2019-01-24 (Paul Pedriana)
  - Initial draft

12.12. XR_KHR_D3D11_enable

Name String

`XR_KHR_D3D11_enable`

Extension Type

Instance extension

Registered Extension Number

28

Revision

5

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2018-11-16

IP Status

No known IP claims.

Contributors

Bryce Hutchings, Microsoft
Paul Pedriana, Oculus
Mark Young, LunarG
Minmin Gong, Microsoft

Overview

This extension enables the use of the D3D11 graphics API in an OpenXR runtime. Without this extension, the OpenXR runtime may not be able to use any D3D11 swapchain images.

This extension provides the mechanisms necessary for an application to generate a valid XrGraphicsBindingD3D11KHR structure in order to create a D3D11-based XrSession. Note that during this process the application is responsible for creating all the required D3D11 objects, including a graphics device to be used for rendering.

This extension also provides mechanisms for the application to interact with images acquired by calling xrEnumerateSwapchainImages.

In order to expose the structures, types, and functions of this extension, you **must** define XR_USE_GRAPHICS_API_D3D11 before including the OpenXR platform header `openxr_platform.h`, in all portions of your library or application that include it.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- XR_TYPE_GRAPHICS_REQUIREMENTS_D3D11_KHR
- XR_TYPE_GRAPHICS_BINDING_D3D11_KHR
- XR_TYPE_SWAPCHAIN_IMAGE_D3D11_KHR

New Enums

New Structures

The following structures are provided to supply supporting runtimes the necessary information required to work with the D3D11 API executing on certain operating systems.

The XrGraphicsBindingD3D11KHR structure is defined as:

```
typedef struct XrGraphicsBindingD3D11KHR {
    XrStructureType    type;
    const void*        next;
    ID3D11Device*      device;
} XrGraphicsBindingD3D11KHR;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- device is a pointer to a valid ID3D11Device to use.

When creating a D3D11-backed XrSession, the application will provide a pointer to an XrGraphicsBindingD3D11KHR in the next chain of the XrSessionCreateInfo.

Valid Usage (Implicit)

- The `XR_KHR_D3D11_enable` extension **must** be enabled prior to using XrGraphicsBindingD3D11KHR
- `type` **must** be `XR_TYPE_GRAPHICS_BINDING_D3D11_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `device` **must** be a pointer to an `ID3D11Device` value

The XrSwapchainImageD3D11KHR structure is defined as:

```
typedef struct XrSwapchainImageD3D11KHR {
     XrStructureType    type;
    void*              next;
    ID3D11Texture2D*    texture;
} XrSwapchainImageD3D11KHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `texture` is a pointer to a valid `ID3D11Texture2D` to use.

If a given session was created with XrGraphicsBindingD3D11KHR, the following conditions **must** apply.

- Calls to xrEnumerateSwapchainImages on an XrSwapchain in that session **must** return an array of XrSwapchainImageD3D11KHR structures.
- Whenever an OpenXR function accepts an XrSwapchainImageBaseHeader pointer as a parameter in that session, the runtime **must** also accept a pointer to an XrSwapchainImageD3D11KHR.

The OpenXR runtime **must** interpret the top-left corner of the swapchain image as the coordinate origin unless specified otherwise by extension functionality.

The OpenXR runtime **must** interpret the swapchain images in a clip space of positive Y pointing up, near Z plane at 0, and far Z plane at 1.

Valid Usage (Implicit)

- The XR_KHR_D3D11_enable extension **must** be enabled prior to using XrSwapchainImageD3D11KHR
- type **must** be XR_TYPE_SWAPCHAIN_IMAGE_D3D11_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- texture **must** be a pointer to an ID3D11Texture2D value

The XrGraphicsRequirementsD3D11KHR structure is defined as:

```
typedef struct XrGraphicsRequirementsD3D11KHR {
    XrStructureType      type;
    void*                next;
    LUID                 adapterLuid;
    D3D_FEATURE_LEVEL    minFeatureLevel;
} XrGraphicsRequirementsD3D11KHR;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- adapterLuid identifies what graphics device needs to be used.
- minFeatureLevel is the minimum feature level that the D3D11 device must be initialized with.

XrGraphicsRequirementsD3D11KHR is populated by xrGetD3D11GraphicsRequirementsKHR.

Valid Usage (Implicit)

- The XR_KHR_D3D11_enable extension **must** be enabled prior to using XrGraphicsRequirementsD3D11KHR
- type **must** be XR_TYPE_GRAPHICS_REQUIREMENTS_D3D11_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- adapterLuid **must** be a valid LUID value
- minFeatureLevel **must** be a valid D3D_FEATURE_LEVEL value

New Functions

Some computer systems may have multiple graphics devices, each of which may have independent external display outputs. XR systems that connect to such graphics devices are typically connected to a single device. Applications need to know what graphics device the XR system is connected to so that they can use that graphics device to generate XR images.

To retrieve the D3D11 feature level and graphics device for an instance and system, call:

```
XrResult xrGetD3D11GraphicsRequirementsKHR(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    XrGraphicsRequirementsD3D11KHR*             graphicsRequirements);
```

Parameter Descriptions

- `instance` is an XrInstance handle previously created with xrCreateInstance.
- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `graphicsRequirements` is the XrGraphicsRequirementsD3D11KHR output structure.

The xrGetD3D11GraphicsRequirementsKHR function identifies to the application what graphics device (Windows LUID) needs to be used and the minimum feature level to use. The runtime **must** return `XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING` (`XR_ERROR_VALIDATION_FAILURE` **may** be returned due to legacy behavior) on calls to xrCreateSession if xrGetD3D11GraphicsRequirementsKHR has not been called for the same `instance` and `systemId`. The LUID and feature level that xrGetD3D11GraphicsRequirementsKHR returns should be used to create the `ID3D11Device` that the application passes to xrCreateSession in the XrGraphicsBindingD3D11KHR.

Valid Usage (Implicit)

- The `XR_KHR_D3D11_enable` extension **must** be enabled prior to calling xrGetD3D11GraphicsRequirementsKHR
- `instance` **must** be a valid XrInstance handle
- `graphicsRequirements` **must** be a pointer to an XrGraphicsRequirementsD3D11KHR structure

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_SYSTEM_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_FUNCTION_UNSUPPORTED

Issues

Version History

- Revision 1, 2018-05-07 (Mark Young)
  - Initial draft
- Revision 2, 2018-06-21 (Bryce Hutchings)
  - Split XR_KHR_D3D_enable into XR_KHR_D3D11_enable
  - Rename and expand xrGetD3DGraphicsDeviceKHR functionality to xrGetD3D11GraphicsRequirementsKHR
- Revision 3, 2018-11-15 (Paul Pedriana)
  - Specified the swapchain texture coordinate origin.
- Revision 4, 2018-11-16 (Minmin Gong)
  - Specified Y direction and Z range in clip space
- Revision 5, 2020-08-06 (Bryce Hutchings)
  - Added new XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING error code

12.13. XR_KHR_D3D12_enable

Name String

XR_KHR_D3D12_enable

Extension Type

Instance extension

Registered Extension Number

29

Revision

7

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2020-03-18

IP Status

No known IP claims.

Contributors

Bryce Hutchings, Microsoft
Paul Pedriana, Oculus
Mark Young, LunarG
Minmin Gong, Microsoft
Dan Ginsburg, Valve

Overview

This extension enables the use of the D3D12 graphics API in an OpenXR runtime. Without this extension, the OpenXR runtime may not be able to use any D3D12 swapchain images.

This extension provides the mechanisms necessary for an application to generate a valid XrGraphicsBindingD3D12KHR structure in order to create a D3D12-based XrSession. Note that during this process the application is responsible for creating all the required D3D12 objects, including a graphics device and queue to be used for rendering.

This extension also provides mechanisms for the application to interact with images acquired by calling xrEnumerateSwapchainImages.

In order to expose the structures, types, and functions of this extension, you **must** define XR_USE_GRAPHICS_API_D3D12 before including the OpenXR platform header `openxr_platform.h`, in all portions of your library or application that include it.

Swapchain Image Resource State

When an application acquires a swapchain image by calling xrAcquireSwapchainImage in a session create using XrGraphicsBindingD3D12KHR, the OpenXR runtime **must** guarantee that:

- The color rendering target image has a resource state match with `D3D12_RESOURCE_STATE_RENDER_TARGET`

- The depth rendering target image has a resource state match with `D3D12_RESOURCE_STATE_DEPTH_WRITE`
- The `ID3D12CommandQueue` specified in XrGraphicsBindingD3D12KHR can write to the image.

When an application releases a swapchain image by calling xrReleaseSwapchainImage, in a session create using XrGraphicsBindingD3D12KHR, the OpenXR runtime **must** interpret the image as:

- Having a resource state match with `D3D12_RESOURCE_STATE_RENDER_TARGET` if the image is a color rendering target
- Having a resource state match with `D3D12_RESOURCE_STATE_DEPTH_WRITE` if the image is a depth rendering target
- Being available for read/write on the `ID3D12CommandQueue` specified in XrGraphicsBindingD3D12KHR.

The application is responsible for transitioning the swapchain image back to the resource state and queue availability that the OpenXR runtime requires. If the image is not in a resource state match with the above specifications the runtime **may** exhibit undefined behaviour.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_GRAPHICS_REQUIREMENTS_D3D12_KHR`
- `XR_TYPE_GRAPHICS_BINDING_D3D12_KHR`
- `XR_TYPE_SWAPCHAIN_IMAGE_D3D12_KHR`

New Enums

New Structures

The following structures are provided to supply supporting runtimes the necessary information required to work with the D3D12 API executing on certain operating systems.

The XrGraphicsBindingD3D12KHR structure is defined as:

```
typedef struct XrGraphicsBindingD3D12KHR {
    XrStructureType        type;
    const void*            next;
    ID3D12Device*          device;
    ID3D12CommandQueue*    queue;
} XrGraphicsBindingD3D12KHR;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- device is a pointer to a valid ID3D12Device to use.
- queue is a pointer to a valid ID3D12CommandQueue to use.

When creating a D3D12-backed XrSession, the application will provide a pointer to an XrGraphicsBindingD3D12KHR in the next chain of the XrSessionCreateInfo.

Valid Usage (Implicit)

- The XR_KHR_D3D12_enable extension **must** be enabled prior to using XrGraphicsBindingD3D12KHR
- type **must** be XR_TYPE_GRAPHICS_BINDING_D3D12_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- device **must** be a pointer to an ID3D12Device value
- queue **must** be a pointer to an ID3D12CommandQueue value

The XrSwapchainImageD3D12KHR structure is defined as:

```
typedef struct XrSwapchainImageD3D12KHR {
     XrStructureType    type;
    void*              next;
    ID3D12Resource*    texture;
} XrSwapchainImageD3D12KHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `texture` is a pointer to a valid `ID3D12Texture2D` to use.

If a given session was created with XrGraphicsBindingD3D12KHR, the following conditions **must** apply.

- Calls to xrEnumerateSwapchainImages on an XrSwapchain in that session **must** return an array of XrSwapchainImageD3D12KHR structures.
- Whenever an OpenXR function accepts an XrSwapchainImageBaseHeader pointer as a parameter in that session, the runtime **must** also accept a pointer to an XrSwapchainImageD3D12KHR.

The OpenXR runtime **must** interpret the top-left corner of the swapchain image as the coordinate origin unless specified otherwise by extension functionality.

The OpenXR runtime **must** interpret the swapchain images in a clip space of positive Y pointing up, near Z plane at 0, and far Z plane at 1.

Valid Usage (Implicit)

- The `XR_KHR_D3D12_enable` extension **must** be enabled prior to using XrSwapchainImageD3D12KHR
- `type` **must** be `XR_TYPE_SWAPCHAIN_IMAGE_D3D12_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `texture` **must** be a pointer to an `ID3D12Resource` value

The XrGraphicsRequirementsD3D12KHR structure is defined as:

```
typedef struct XrGraphicsRequirementsD3D12KHR {
    XrStructureType      type;
    void*                next;
    LUID                 adapterLuid;
    D3D_FEATURE_LEVEL    minFeatureLevel;
} XrGraphicsRequirementsD3D12KHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `adapterLuid` identifies what graphics device needs to be used.
- `minFeatureLevel` is the minimum feature level that the D3D12 device must be initialized with.

XrGraphicsRequirementsD3D12KHR is populated by xrGetD3D12GraphicsRequirementsKHR.

Valid Usage (Implicit)

- The `XR_KHR_D3D12_enable` extension **must** be enabled prior to using XrGraphicsRequirementsD3D12KHR
- `type` **must** be `XR_TYPE_GRAPHICS_REQUIREMENTS_D3D12_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `adapterLuid` **must** be a valid `LUID` value
- `minFeatureLevel` **must** be a valid `D3D_FEATURE_LEVEL` value

New Functions

Some computer systems may have multiple graphics devices, each of which may have independent external display outputs. XR systems that connect to such graphics devices are typically connected to a single device. Applications need to know what graphics device the XR system is connected to so that they can use that graphics device to generate XR images.

To retrieve the D3D12 feature level and graphics device for an instance and system, call:

```
XrResult xrGetD3D12GraphicsRequirementsKHR(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    XrGraphicsRequirementsD3D12KHR*             graphicsRequirements);
```

Parameter Descriptions

- `instance` is an XrInstance handle previously created with xrCreateInstance.
- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `graphicsRequirements` is the XrGraphicsRequirementsD3D12KHR output structure.

The xrGetD3D12GraphicsRequirementsKHR function identifies to the application what graphics device (Windows LUID) needs to be used and the minimum feature level to use. The runtime **must** return `XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING` (`XR_ERROR_VALIDATION_FAILURE` **may** be returned due to legacy behavior) on calls to xrCreateSession if xrGetD3D12GraphicsRequirementsKHR has not been called for the same `instance` and `systemId`. The LUID and feature level that xrGetD3D12GraphicsRequirementsKHR returns should be used to create the `ID3D12Device` that the application passes to xrCreateSession in the XrGraphicsBindingD3D12KHR.

Valid Usage (Implicit)

- The `XR_KHR_D3D12_enable` extension **must** be enabled prior to calling xrGetD3D12GraphicsRequirementsKHR
- `instance` **must** be a valid XrInstance handle
- `graphicsRequirements` **must** be a pointer to an XrGraphicsRequirementsD3D12KHR structure

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_SYSTEM_INVALID`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_FUNCTION_UNSUPPORTED`

Issues

Version History

- Revision 1, 2018-05-07 (Mark Young)
  - Initial draft
- Revision 2, 2018-06-21 (Bryce Hutchings)
  - Split `XR_KHR_D3D_enable` into `XR_KHR_D3D12_enable`
  - Rename and expand `xrGetD3DGraphicsDeviceKHR` functionality to `xrGetD3D12GraphicsRequirementsKHR`
- Revision 3, 2018-11-15 (Paul Pedriana)
  - Specified the swapchain texture coordinate origin.
- Revision 4, 2018-11-16 (Minmin Gong)
  - Specified Y direction and Z range in clip space
- Revision 5, 2019-01-29 (Dan Ginsburg)
  - Added swapchain image resource state details.
- Revision 6, 2020-03-18 (Minmin Gong)
  - Specified depth swapchain image resource state.
- Revision 7, 2020-08-06 (Bryce Hutchings)
  - Added new `XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING` error code

12.14. XR_KHR_loader_init

Name String

`XR_KHR_loader_init`

Extension Type

Instance extension

Registered Extension Number

89

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2020-05-07

IP Status

No known IP claims.

Contributors

Cass Everitt, Facebook

Overview

On some platforms, before loading can occur the loader must be initialized with platform-specific parameters. Unlike other extensions, the presence of this extension is signaled by a successful call to xrGetInstanceProcAddr to retrieve the function pointer for xrInitializeLoaderKHR using a null instance handle. If this extension is supported, its use may be required on some platforms and the use of the xrInitializeLoaderKHR function must precede other OpenXR calls except xrGetInstanceProcAddr. This function exists as part of the loader library that the application is using.

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

The XrLoaderInitInfoBaseHeaderKHR structure is defined as:

```
typedef struct XrLoaderInitInfoBaseHeaderKHR {
    XrStructureType    type;
    const void*        next;
} XrLoaderInitInfoBaseHeaderKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure. This base structure itself has no associated XrStructureType value.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.

Valid Usage (Implicit)

- The XR_KHR_loader_init extension **must** be enabled prior to using XrLoaderInitInfoBaseHeaderKHR
- type **must** be XR_TYPE_LOADER_INIT_INFO_ANDROID_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain

New Functions

To initialize an OpenXR loader with platform or implementation-specific parameters, call:

```
XrResult xrInitializeLoaderKHR(
    const XrLoaderInitInfoBaseHeaderKHR*        loaderInitInfo);
```

Parameter Descriptions

- loaderInitInfo is a pointer to an XrLoaderInitInfoBaseHeaderKHR structure, which is a polymorphic type defined by other platform- or implementation-specific extensions.

Issues

Version History

- Revision 1, 2020-05-07 (Cass Everitt)
  - Initial draft

12.15. XR_KHR_loader_init_android

Name String

XR_KHR_loader_init_android

Extension Type

Instance extension

Registered Extension Number

90

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0
- Requires `XR_KHR_loader_init`

Last Modified Date

2020-05-07

IP Status

No known IP claims.

Contributors

Cass Everitt, Facebook

Overview

On Android, some loader implementations need the application to provide additional information on initialization. This extension defines the parameters needed by such implementations. If this is available on a given implementation, an application **must** make use of it.

On implementations where use of this is required, the following condition **must** apply:

- Whenever an OpenXR function accepts an XrLoaderInitInfoBaseHeaderKHR pointer, the runtime (and loader) **must** also accept a pointer to an XrLoaderInitInfoAndroidKHR.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_LOADER_INIT_INFO_ANDROID_KHR`

New Enums

New Structures

The XrLoaderInitInfoAndroidKHR structure is defined as:

```
typedef struct XrLoaderInitInfoAndroidKHR {
    XrStructureType    type;
    const void*        next;
    void*              applicationVM;
    void*              applicationContext;
} XrLoaderInitInfoAndroidKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `applicationVM` is a pointer to the JNI's opaque `JavaVM` structure, cast to a void pointer.
- `applicationContext` is a JNI reference to an `android.content.Context` associated with the application, cast to a void pointer.

Valid Usage (Implicit)

- The `XR_KHR_loader_init_android` extension **must** be enabled prior to using XrLoaderInitInfoAndroidKHR
- `type` **must** be `XR_TYPE_LOADER_INIT_INFO_ANDROID_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `applicationVM` **must** be a pointer value
- `applicationContext` **must** be a pointer value

New Functions

Issues

Version History

- Revision 1, 2020-05-07 (Cass Everitt)
  - Initial draft

12.16. XR_KHR_opengl_enable

Name String

`XR_KHR_opengl_enable`

Extension Type

Instance extension

Registered Extension Number

24

Revision

9

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2019-07-02

IP Status

No known IP claims.

Contributors

Mark Young, LunarG
Bryce Hutchings, Microsoft
Paul Pedriana, Oculus
Minmin Gong, Microsoft
Robert Menzel, NVIDIA
Jakob Bornecrantz, Collabora

Overview

This extension enables the use of the OpenGL graphics API in an OpenXR runtime. Without this extension, the OpenXR runtime **may** not be able to provide any OpenGL swapchain images.

This extension provides the mechanisms necessary for an application to generate a valid `XrGraphicsBindingOpenGL*KHR` structure in order to create an OpenGL-based XrSession. Note that during this process the application is responsible for creating an OpenGL context to be used for rendering. The runtime however will provide the OpenGL textures to render into in the form of a swapchain.

This extension provides mechanisms for the application to interact with images acquired by calling xrEnumerateSwapchainImages.

In order to expose the structures, types, and functions of this extension, the application **must** define XR_USE_GRAPHICS_API_OPENGL, as well as an appropriate window system define supported by this extension, before including the OpenXR platform header `openxr_platform.h`, in all portions of the library or application that include it. The window system defines currently supported by this extension are:

- XR_USE_PLATFORM_WIN32

* XR_USE_PLATFORM_XLIB
* XR_USE_PLATFORM_XCB
* XR_USE_PLATFORM_WAYLAND

Note that a runtime implementation of this extension is only required to support the structs introduced by this extension which belong to the platform it is running on.

Note that the OpenGL context given to the call xrCreateSession **must** not be bound in another thread when calling the functions: xrCreateSession, xrDestroySession, xrBeginFrame, xrEndFrame, xrCreateSwapchain, xrDestroySwapchain, xrEnumerateSwapchainImages, xrAcquireSwapchainImage, xrWaitSwapchainImage and xrReleaseSwapchainImage. It **may** be bound in the thread calling those functions. The runtime **must** not access the context from any other function. In particular the application must be able to call xrWaitFrame from a different thread than the rendering thread.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

* `XR_TYPE_GRAPHICS_REQUIREMENTS_OPENGL_KHR`
* `XR_TYPE_GRAPHICS_BINDING_OPENGL_WIN32_KHR`
* `XR_TYPE_GRAPHICS_BINDING_OPENGL_XLIB_KHR`
* `XR_TYPE_GRAPHICS_BINDING_OPENGL_XCB_KHR`
* `XR_TYPE_GRAPHICS_BINDING_OPENGL_WAYLAND_KHR`
* `XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_KHR`

New Enums

New Structures

The following structures are provided to supply supporting runtimes the necessary information required to work with the OpenGL API executing on certain operating systems.

These structures are only available when the corresponding `XR_USE_PLATFORM_` macro is defined before including `openxr_platform.h`.

The XrGraphicsBindingOpenGLWin32KHR structure is defined as:

```
typedef struct XrGraphicsBindingOpenGLWin32KHR {
    XrStructureType    type;
    const void*        next;
    HDC                hDC;
    HGLRC              hGLRC;
} XrGraphicsBindingOpenGLWin32KHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `hDC` is a valid Windows HW device context handle.
- `hGLRC` is a valid Windows OpenGL rendering context handle.

When creating an OpenGL-backed XrSession on Microsoft Windows, the application will provide a pointer to an XrGraphicsBindingOpenGLWin32KHR in the `next` chain of the XrSessionCreateInfo. As no standardized way exists for OpenGL to create the graphics context on a specific GPU, the runtime **must** assume that the application uses the operating systems default GPU. If the GPU used by the runtime does not match the GPU on which the OpenGL context of the application got created, xrCreateSession **must** return `XR_ERROR_GRAPHICS_DEVICE_INVALID`.

The required window system configuration define to expose this structure type is XR_USE_PLATFORM_WIN32.

Valid Usage (Implicit)

- The `XR_KHR_opengl_enable` extension **must** be enabled prior to using XrGraphicsBindingOpenGLWin32KHR
- `type` **must** be `XR_TYPE_GRAPHICS_BINDING_OPENGL_WIN32_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `hDC` **must** be a valid `HDC` value
- `hGLRC` **must** be a valid `HGLRC` value

The XrGraphicsBindingOpenGLXlibKHR structure is defined as:

```
typedef struct XrGraphicsBindingOpenGLXlibKHR {
    XrStructureType    type;
    const void*        next;
    Display*           xDisplay;
    uint32_t           visualid;
    GLXFBConfig        glxFBConfig;
    GLXDrawable        glxDrawable;
    GLXContext         glxContext;
} XrGraphicsBindingOpenGLXlibKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `xDisplay` is a valid X11 `Display`.
- `visualid` is a valid X11 visual identifier.
- `glxFBConfig` is a valid X11 OpenGL GLX `GLXFBConfig`.
- `glxDrawable` is a valid X11 OpenGL GLX `GLXDrawable`.
- `glxContext` is a valid X11 OpenGL GLX `GLXContext`.

When creating an OpenGL-backed XrSession on any Linux/Unix platform that utilizes X11 and GLX, via the Xlib library, the application will provide a pointer to an XrGraphicsBindingOpenGLXlibKHR in the `next` chain of the XrSessionCreateInfo.

The required window system configuration define to expose this structure type is XR_USE_PLATFORM_XLIB.

Valid Usage (Implicit)

- The XR_KHR_opengl_enable extension **must** be enabled prior to using XrGraphicsBindingOpenGLXlibKHR
- type **must** be XR_TYPE_GRAPHICS_BINDING_OPENGL_XLIB_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- xDisplay **must** be a pointer to a Display value
- glxFBConfig **must** be a valid GLXFBConfig value
- glxDrawable **must** be a valid GLXDrawable value
- glxContext **must** be a valid GLXContext value

The XrGraphicsBindingOpenGLXcbKHR structure is defined as:

```
typedef struct XrGraphicsBindingOpenGLXcbKHR {
    XrStructureType       type;
    const void*           next;
    xcb_connection_t*     connection;
    uint32_t              screenNumber;
    xcb_glx_fbconfig_t    fbconfigid;
    xcb_visualid_t        visualid;
    xcb_glx_drawable_t    glxDrawable;
    xcb_glx_context_t     glxContext;
} XrGraphicsBindingOpenGLXcbKHR;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- connection is a valid xcb_connection_t.
- screenNumber is an index indicating which screen should be used for rendering.
- fbconfigid is a valid XCB OpenGL GLX xcb_glx_fbconfig_t.
- visualid is a valid XCB OpenGL GLX xcb_visualid_t.
- glxDrawable is a valid XCB OpenGL GLX xcb_glx_drawable_t.
- glxContext is a valid XCB OpenGL GLX xcb_glx_context_t.

When creating an OpenGL-backed XrSession on any Linux/Unix platform that utilizes X11 and GLX, via the Xlib library, the application will provide a pointer to an XrGraphicsBindingOpenGLXcbKHR in the next chain of the XrSessionCreateInfo.

The required window system configuration define to expose this structure type is XR_USE_PLATFORM_XCB.

Valid Usage (Implicit)

- The XR_KHR_opengl_enable extension **must** be enabled prior to using XrGraphicsBindingOpenGLXcbKHR
- type **must** be XR_TYPE_GRAPHICS_BINDING_OPENGL_XCB_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- connection **must** be a pointer to an xcb_connection_t value
- fbconfigid **must** be a valid xcb_glx_fbconfig_t value
- visualid **must** be a valid xcb_visualid_t value
- glxDrawable **must** be a valid xcb_glx_drawable_t value
- glxContext **must** be a valid xcb_glx_context_t value

The XrGraphicsBindingOpenGLWaylandKHR structure is defined as:

```
typedef struct XrGraphicsBindingOpenGLWaylandKHR {
    XrStructureType       type;
    const void*           next;
    struct wl_display*    display;
} XrGraphicsBindingOpenGLWaylandKHR;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- display is a valid Wayland wl_display.

When creating an OpenGL-backed XrSession on any Linux/Unix platform that utilizes the Wayland protocol with its compositor, the application will provide a pointer to an XrGraphicsBindingOpenGLWaylandKHR in the next chain of the XrSessionCreateInfo.

The required window system configuration define to expose this structure type is XR_USE_PLATFORM_WAYLAND.

Valid Usage (Implicit)

- The XR_KHR_opengl_enable extension **must** be enabled prior to using XrGraphicsBindingOpenGLWaylandKHR
- type **must** be XR_TYPE_GRAPHICS_BINDING_OPENGL_WAYLAND_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- display **must** be a pointer to a wl_display value

The XrSwapchainImageOpenGLKHR structure is defined as:

```
typedef struct XrSwapchainImageOpenGLKHR {
    XrStructureType    type;
    void*              next;
    uint32_t           image;
} XrSwapchainImageOpenGLKHR;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- image is the OpenGL texture handle associated with this swapchain image.

If a given session was created with a XrGraphicsBindingOpenGL*KHR, the following conditions **must** apply.

- Calls to xrEnumerateSwapchainImages on an XrSwapchain in that session **must** return an array of XrSwapchainImageOpenGLKHR structures.
- Whenever an OpenXR function accepts an XrSwapchainImageBaseHeader pointer as a parameter in that session, the runtime **must** also accept a pointer to an XrSwapchainImageOpenGLKHR.

The OpenXR runtime **must** interpret the bottom-left corner of the swapchain image as the coordinate origin unless specified otherwise by extension functionality.

The OpenXR runtime **must** interpret the swapchain images in a clip space of positive Y pointing up, near Z plane at -1, and far Z plane at 1.

Valid Usage (Implicit)

- The XR_KHR_opengl_enable extension **must** be enabled prior to using XrSwapchainImageOpenGLKHR
- type **must** be XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain

The XrGraphicsRequirementsOpenGLKHR structure is defined as:

```
typedef struct XrGraphicsRequirementsOpenGLKHR {
    XrStructureType    type;
    void*              next;
    XrVersion          minApiVersionSupported;
    XrVersion          maxApiVersionSupported;
} XrGraphicsRequirementsOpenGLKHR;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- minApiVersionSupported is the minimum version of OpenGL that the runtime supports. Uses XR_MAKE_VERSION on major and minor API version, ignoring any patch version component.
- maxApiVersionSupported is the maximum version of OpenGL that the runtime has been tested on and is known to support. Newer OpenGL versions might work if they are compatible. Uses XR_MAKE_VERSION on major and minor API version, ignoring any patch version component.

XrGraphicsRequirementsOpenGLKHR is populated by xrGetOpenGLGraphicsRequirementsKHR with the runtime's OpenGL API version requirements.

Valid Usage (Implicit)

- The XR_KHR_opengl_enable extension **must** be enabled prior to using XrGraphicsRequirementsOpenGLKHR
- type **must** be XR_TYPE_GRAPHICS_REQUIREMENTS_OPENGL_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain

New Functions

To query OpenGL API version requirements for an instance and system, call:

```
XrResult xrGetOpenGLGraphicsRequirementsKHR(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    XrGraphicsRequirementsOpenGLKHR*            graphicsRequirements);
```

Parameter Descriptions

- `instance` is an XrInstance handle previously created with xrCreateInstance.
- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `graphicsRequirements` is the XrGraphicsRequirementsOpenGLKHR output structure.

The xrGetOpenGLGraphicsRequirementsKHR function identifies to the application the minimum OpenGL version requirement and the highest known tested OpenGL version. The runtime **must** return `XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING` (`XR_ERROR_VALIDATION_FAILURE` **may** be returned due to legacy behavior) on calls to xrCreateSession if xrGetOpenGLGraphicsRequirementsKHR has not been called for the same `instance` and `systemId`.

Valid Usage (Implicit)

- The `XR_KHR_opengl_enable` extension **must** be enabled prior to calling xrGetOpenGLGraphicsRequirementsKHR
- `instance` **must** be a valid XrInstance handle
- `graphicsRequirements` **must** be a pointer to an XrGraphicsRequirementsOpenGLKHR structure

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_SYSTEM_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_FUNCTION_UNSUPPORTED

Issues

Version History

- Revision 1, 2018-05-07 (Mark Young)
  - Initial draft
- Revision 2, 2018-06-21 (Bryce Hutchings)
  - Add new xrGetOpenGLGraphicsRequirementsKHR
- Revision 3, 2018-11-15 (Paul Pedriana)
  - Specified the swapchain texture coordinate origin.
- Revision 4, 2018-11-16 (Minmin Gong)
  - Specified Y direction and Z range in clip space
- Revision 5, 2019-01-25 (Robert Menzel)
  - Description updated
- Revision 6, 2019-07-02 (Robert Menzel)
  - Minor fixes
- Revision 7, 2019-07-08 (Ryan Pavlk)
  - Adjusted member name in XCB struct
- Revision 8, 2019-11-28 (Jakob Bornecrantz)
  - Added note about context not allowed to be current in a different thread.
- Revision 9, 2020-08-06 (Bryce Hutchings)
  - Added new XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING error code

12.17. XR_KHR_opengl_es_enable

Name String

`XR_KHR_opengl_es_enable`

Extension Type

Instance extension

Registered Extension Number

25

Revision

7

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2019-07-12

IP Status

No known IP claims.

Contributors

Mark Young, LunarG
Bryce Hutchings, Microsoft
Paul Pedriana, Oculus
Minmin Gong, Microsoft
Robert Menzel, NVIDIA
Martin Renschler, Qualcomm

Overview

This extension must be provided by runtimes supporting applications using OpenGL ES APIs for rendering. OpenGL ES applications need this extension to obtain compatible swapchain images which the runtime is required to supply. The runtime needs the following OpenGL ES objects from the application in order to interact properly with the OpenGL ES driver: EGLDisplay, EGLConfig and EGLContext.

These are passed from the application to the runtime in a XrGraphicsBindingOpenGLESAndroidKHR structure when creating the XrSession. Although not restricted to Android, the OpenGL ES extension is currently tailored for Android.

Note that the application is responsible for creating the required OpenGL ES objects, including an OpenGL ES context to be used for rendering.

This extension also provides mechanisms for the application to interact with images acquired by calling xrEnumerateSwapchainImages.

In order to expose the structures, types, and functions of this extension, the application source code **must** define XR_USE_GRAPHICS_API_OPENGL_ES, as well as an appropriate window system define, before including the OpenXR platform header `openxr_platform.h`, in all portions of your library or application that include it. The only window system define currently supported by this extension is:

- XR_USE_PLATFORM_ANDROID

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_GRAPHICS_REQUIREMENTS_OPENGL_ES_KHR`
- `XR_TYPE_GRAPHICS_BINDING_OPENGL_ES_ANDROID_KHR`
- `XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_ES_KHR`

New Enums

New Structures

The following structures are provided to supply supporting runtimes the necessary information required to work with the OpenGL ES API executing on certain operating systems.

These structures are only available when the corresponding `XR_USE_PLATFORM_` macro is defined before including `openxr_platform.h`.

The XrGraphicsBindingOpenGLESAndroidKHR structure is defined as:

```
typedef struct XrGraphicsBindingOpenGLESAndroidKHR {
    XrStructureType    type;
    const void*        next;
    EGLDisplay         display;
    EGLConfig          config;
    EGLContext         context;
} XrGraphicsBindingOpenGLESAndroidKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `display` is a valid Android OpenGL ES `EGLDisplay`.
- `config` is a valid Android OpenGL ES `EGLConfig`.
- `context` is a valid Android OpenGL ES `EGLContext`.

When creating an OpenGL ES-backed XrSession on Android, the application will provide a pointer to an XrGraphicsBindingOpenGLESAndroidKHR structure in the `next` chain of the XrSessionCreateInfo.

The required window system configuration define to expose this structure type is XR_USE_PLATFORM_ANDROID.

Valid Usage (Implicit)

- The `XR_KHR_opengl_es_enable` extension **must** be enabled prior to using XrGraphicsBindingOpenGLESAndroidKHR
- `type` **must** be `XR_TYPE_GRAPHICS_BINDING_OPENGL_ES_ANDROID_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `display` **must** be a valid `EGLDisplay` value
- `config` **must** be a valid `EGLConfig` value
- `context` **must** be a valid `EGLContext` value

The XrSwapchainImageOpenGLESKHR structure is defined as:

```
typedef struct XrSwapchainImageOpenGLESKHR {
    XrStructureType    type;
    void*              next;
    uint32_t           image;
} XrSwapchainImageOpenGLESKHR;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- image is an index indicating the current OpenGL ES swapchain image to use.

If a given session was created with a XrGraphicsBindingOpenGLES*KHR, the following conditions **must** apply.

- Calls to xrEnumerateSwapchainImages on an XrSwapchain in that session **must** return an array of XrSwapchainImageOpenGLESKHR structures.
- Whenever an OpenXR function accepts an XrSwapchainImageBaseHeader pointer as a parameter in that session, the runtime **must** also accept a pointer to an XrSwapchainImageOpenGLESKHR structure.

The OpenXR runtime **must** interpret the bottom-left corner of the swapchain image as the coordinate origin unless specified otherwise by extension functionality.

The OpenXR runtime **must** interpret the swapchain images in a clip space of positive Y pointing up, near Z plane at -1, and far Z plane at 1.

Valid Usage (Implicit)

- The XR_KHR_opengl_es_enable extension **must** be enabled prior to using XrSwapchainImageOpenGLESKHR
- type **must** be XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_ES_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain

The XrGraphicsRequirementsOpenGLESKHR structure is defined as:

```
typedef struct XrGraphicsRequirementsOpenGLESKHR {
    XrStructureType    type;
    void*              next;
    XrVersion          minApiVersionSupported;
    XrVersion          maxApiVersionSupported;
} XrGraphicsRequirementsOpenGLESKHR;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- minApiVersionSupported is the minimum version of OpenGL ES that the runtime supports. Uses XR_MAKE_VERSION on major and minor API version, ignoring any patch version component.
- maxApiVersionSupported is the maximum version of OpenGL ES that the runtime has been tested on and is known to support. Newer OpenGL ES versions might work if they are compatible. Uses XR_MAKE_VERSION on major and minor API version, ignoring any patch version component.

XrGraphicsRequirementsOpenGLESKHR is populated by xrGetOpenGLESGraphicsRequirementsKHR with the runtime's OpenGL ES API version requirements.

Valid Usage (Implicit)

- The XR_KHR_opengl_es_enable extension **must** be enabled prior to using XrGraphicsRequirementsOpenGLESKHR
- type **must** be XR_TYPE_GRAPHICS_REQUIREMENTS_OPENGL_ES_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain

New Functions

To query OpenGL ES API version requirements for an instance and system, call:

```
XrResult xrGetOpenGLESGraphicsRequirementsKHR(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    XrGraphicsRequirementsOpenGLESKHR*          graphicsRequirements);
```

Parameter Descriptions

- `instance` is an XrInstance handle previously created with xrCreateInstance.
- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `graphicsRequirements` is the XrGraphicsRequirementsOpenGLESKHR output structure.

The xrGetOpenGLESGraphicsRequirementsKHR function identifies to the application the minimum OpenGL ES version requirement and the highest known tested OpenGL ES version. The runtime **must** return `XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING` (`XR_ERROR_VALIDATION_FAILURE` **may** be returned due to legacy behavior) on calls to xrCreateSession if xrGetOpenGLESGraphicsRequirementsKHR has not been called for the same `instance` and `systemId`.

Valid Usage (Implicit)

- The `XR_KHR_opengl_es_enable` extension **must** be enabled prior to calling xrGetOpenGLESGraphicsRequirementsKHR
- `instance` **must** be a valid XrInstance handle
- `graphicsRequirements` **must** be a pointer to an XrGraphicsRequirementsOpenGLESKHR structure

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_SYSTEM_INVALID`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_FUNCTION_UNSUPPORTED`

Issues

Version History

- Revision 1, 2018-05-07 (Mark Young)

    - Initial draft
- Revision 2, 2018-06-21 (Bryce Hutchings)
    - Add new xrGetOpenGLESGraphicsRequirementsKHR
- Revision 3, 2018-11-15 (Paul Pedriana)
    - Specified the swapchain texture coordinate origin.
- Revision 4, 2018-11-16 (Minmin Gong)
    - Specified Y direction and Z range in clip space
- Revision 5, 2019-01-25 (Robert Menzel)
    - Description updated
- Revision 6, 2019-07-12 (Martin Renschler)
    - Description updated
- Revision 7, 2020-08-06 (Bryce Hutchings)
    - Added new XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING error code

12.18. XR_KHR_visibility_mask

Name String

XR_KHR_visibility_mask

Extension Type

Instance extension

Registered Extension Number

32

Revision

2

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2018-07-05

IP Status

No known IP claims.

Contributors

Paul Pedriana, Oculus
Alex Turner, Microsoft

Contacts

Paul Pedriana, Oculus

Overview

This extension support the providing of a per-view drawing mask for applications. The primary purpose of this is to enable performance improvements that result from avoiding drawing on areas that aren't visible to the user. A common occurrence in head-mounted VR hardware is that the optical system's frustum doesn't intersect precisely with the rectangular display it is viewing. As a result, it may be that there are parts of the display that aren't visible to the user, such as the corners of the display. In such cases it would be unnecessary for the application to draw into those parts.

New Object Types

New Flag Types

New Enum Constants

New Enums

XrVisibilityMaskTypeKHR identifies the different types of mask specification that is supported. The application **can** request a view mask in any of the formats identified by these types.

```
typedef enum XrVisibilityMaskTypeKHR {
    XR_VISIBILITY_MASK_TYPE_HIDDEN_TRIANGLE_MESH_KHR = 1,
    XR_VISIBILITY_MASK_TYPE_VISIBLE_TRIANGLE_MESH_KHR = 2,
    XR_VISIBILITY_MASK_TYPE_LINE_LOOP_KHR = 3,
    XR_VISIBILITY_MASK_TYPE_MAX_ENUM_KHR = 0x7FFFFFFF
} XrVisibilityMaskTypeKHR;
```

Enumerant Descriptions

- XR_VISIBILITY_MASK_TYPE_HIDDEN_TRIANGLE_MESH_KHR refers to a two dimensional triangle mesh on the view surface which **should** not be drawn to by the application. XrVisibilityMaskKHR refers to a set of triangles identified by vertices and vertex indices. The index count will thus be a multiple of three. The triangle vertices will be returned in counter-clockwise order as viewed from the user perspective.
- XR_VISIBILITY_MASK_TYPE_VISIBLE_TRIANGLE_MESH_KHR refers to a two dimensional triangle mesh on the view surface which **should** be drawn to by the application. XrVisibilityMaskKHR refers to a set of triangles identified by vertices and vertex indices. The index count will thus be a multiple of three. The triangle vertices will be returned in counter-clockwise order as viewed from the user perspective.
- XR_VISIBILITY_MASK_TYPE_LINE_LOOP_KHR refers to a single multi-segmented line loop on the view surface which encompasses the view area which **should** be drawn by the application. It is the border that exists between the visible and hidden meshes identified by XR_VISIBILITY_MASK_TYPE_HIDDEN_TRIANGLE_MESH_KHR and XR_VISIBILITY_MASK_TYPE_VISIBLE_TRIANGLE_MESH_KHR. The line is counter-clockwise, contiguous, and non-self crossing, with the last point implicitly connecting to the first point. There is one vertex per point, the index count will equal the vertex count, and the indices will refer to the vertices.

New Structures

The XrVisibilityMaskKHR structure is an input/output struct which specifies the view mask.

```
typedef struct XrVisibilityMaskKHR {
    XrStructureType    type;
    void*              next;
    uint32_t           vertexCapacityInput;
    uint32_t           vertexCountOutput;
    XrVector2f*        vertices;
    uint32_t           indexCapacityInput;
    uint32_t           indexCountOutput;
    uint32_t*          indices;
} XrVisibilityMaskKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `vertexCapacityInput` is the capacity of the `vertices` array, or `0` to indicate a request to retrieve the required capacity.
- `vertexCountOutput` is filled in by the runtime with the count of vertices written or the required capacity in the case that `vertexCapacityInput` or `indexCapacityInput` is `0`.
- `vertices` is an array of vertices filled in by the runtime that specifies mask coordinates in the z=-1 plane of the rendered view—i.e. one meter in front of the view. When rendering the mask for use in a projection layer, these vertices must be transformed by the application's projection matrix used for the respective XrCompositionLayerProjectionView.
- `indexCapacityInput` is the capacity of the `indices` array, or `0` to indicate a request to retrieve the required capacity.
- `indexCountOutput` is filled in by the runtime with the count of indices written or the required capacity in the case that `vertexCapacityInput` or `indexCapacityInput` is `0`.
- `indices` is an array of indices filled in by the runtime, specifying the indices of the mask geometry in the `vertices` array.

Valid Usage (Implicit)

- The `XR_KHR_visibility_mask` extension **must** be enabled prior to using XrVisibilityMaskKHR
- `type` **must** be `XR_TYPE_VISIBILITY_MASK_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- If `vertexCapacityInput` is not `0`, `vertices` **must** be a pointer to an array of `vertexCapacityInput` XrVector2f structures
- If `indexCapacityInput` is not `0`, `indices` **must** be a pointer to an array of `indexCapacityInput` `uint32_t` values

The XrEventDataVisibilityMaskChangedKHR structure specifies an event which indicates that a given view mask has changed. The application **should** respond to the event by calling xrGetVisibilityMaskKHR to retrieve the updated mask. This event is per-view, so if the masks for multiple views in a configuration change then multiple instances of this event will be sent to the application, one per view.

```
typedef struct XrEventDataVisibilityMaskChangedKHR {
    XrStructureType            type;
    const void*                next;
    XrSession                  session;
    XrViewConfigurationType    viewConfigurationType;
    uint32_t                   viewIndex;
} XrEventDataVisibilityMaskChangedKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `session` is the XrSession for which the view mask has changed.
- `viewConfigurationType` is the view configuration whose mask has changed.
- `viewIndex` is the individual view within the view configuration to which the change refers.

Valid Usage (Implicit)

- The `XR_KHR_visibility_mask` extension **must** be enabled prior to using XrEventDataVisibilityMaskChangedKHR
- `type` **must** be `XR_TYPE_EVENT_DATA_VISIBILITY_MASK_CHANGED_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `session` **must** be a valid XrSession handle
- `viewConfigurationType` **must** be a valid XrViewConfigurationType value

New Functions

The xrGetVisibilityMaskKHR function is defined as:

```
XrResult xrGetVisibilityMaskKHR(
    XrSession                                   session,
    XrViewConfigurationType                     viewConfigurationType,
    uint32_t                                    viewIndex,
    XrVisibilityMaskTypeKHR                     visibilityMaskType,
    XrVisibilityMaskKHR*                        visibilityMask);
```

Parameter Descriptions

- `session` is an XrSession handle previously created with xrCreateSession.
- `viewConfigurationType` is the view configuration from which to retrieve mask information.
- `viewIndex` is the individual view within the view configuration from which to retrieve mask information.
- `visibilityMaskType` is the type of visibility mask requested.
- `visibilityMask` is an input/output struct which specifies the view mask.

xrGetVisibilityMaskKHR retrieves the view mask for a given view. This function follows the two-call idiom for filling multiple buffers in a struct. Specifically, if either `vertexCapacityInput` or `indexCapacityInput` is `0`, the runtime **must** respond as if both fields were set to `0`, returning the vertex count and index count through `vertexCountOutput` or `indexCountOutput` respectively. If a view mask for the specified view isn't available, the returned vertex and index counts **must** be `0`.

Valid Usage (Implicit)

- The `XR_KHR_visibility_mask` extension **must** be enabled prior to calling xrGetVisibilityMaskKHR
- `session` **must** be a valid XrSession handle
- `viewConfigurationType` **must** be a valid XrViewConfigurationType value
- `visibilityMaskType` **must** be a valid XrVisibilityMaskTypeKHR value
- `visibilityMask` **must** be a pointer to an XrVisibilityMaskKHR structure

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED`
- `XR_ERROR_SIZE_INSUFFICIENT`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_FUNCTION_UNSUPPORTED`

Issues

Version History

- Revision 1, 2018-07-05 (Paul Pedriana)
  - Initial version.
- Revision 2, 2019-07-15 (Alex Turner)
  - Adjust two-call idiom usage.

12.19. XR_KHR_vulkan_enable

Name String

`XR_KHR_vulkan_enable`

Extension Type

Instance extension

Registered Extension Number

26

Revision

8

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2019-01-25

IP Status

No known IP claims.

Contributors

Mark Young, LunarG
Paul Pedriana, Oculus
Ed Hutchins, Oculus
Andres Rodriguez, Valve
Dan Ginsburg, Valve
Bryce Hutchings, Microsoft
Minmin Gong, Microsoft
Robert Menzel, NVIDIA
Paulo F. Gomes, Samsung Electronics

Overview

This extension enables the use of the Vulkan graphics API in an OpenXR runtime. Without this extension, the OpenXR runtime may not be able to use any Vulkan swapchain images.

This extension provides the mechanisms necessary for an application to generate a valid XrGraphicsBindingVulkanKHR structure in order to create a Vulkan-based XrSession. Note that during this process the application is responsible for creating all the required Vulkan objects.

This extension also provides mechanisms for the application to interact with images acquired by calling xrEnumerateSwapchainImages.

In order to expose the structures, types, and functions of this extension, you **must** define XR_USE_GRAPHICS_API_VULKAN before including the OpenXR platform header `openxr_platform.h`, in all portions of your library or application that include it.

Initialization

Some of the requirements for creating a valid XrGraphicsBindingVulkanKHR include correct initialization of a `VkInstance`, `VkPhysicalDevice`, and `VkDevice`.

A runtime **may** require that the `VkInstance` be initialized to a specific Vulkan API version. Additionally, the runtime **may** require a set of instance extensions to be enabled in the `VkInstance`. These requirements can be queried by the application using xrGetVulkanGraphicsRequirementsKHR and xrGetVulkanInstanceExtensionsKHR, respectively.

Similarly, the runtime **may** require the `VkDevice` to have a set of device extensions enabled, which can be queried using xrGetVulkanDeviceExtensionsKHR.

In order to satisfy the `VkPhysicalDevice` requirements, the application can query xrGetVulkanGraphicsDeviceKHR to identify the correct `VkPhysicalDevice`.

Populating an XrGraphicsBindingVulkanKHR with a `VkInstance`, `VkDevice`, or `VkPhysicalDevice` that does not meet the requirements outlined by this extension **may** result in undefined behaviour by the OpenXR runtime.

The API version, instance extension, device extension and physical device requirements only apply to the `VkInstance`, `VkDevice`, and `VkPhysicalDevice` objects which the application wishes to associate with an XrGraphicsBindingVulkanKHR.

Concurrency

Vulkan requires that concurrent access to a `VkQueue` from multiple threads be externally synchronized. Therefore, OpenXR functions that may access the `VkQueue` specified in the XrGraphicsBindingVulkanKHR must also be externally synchronized.

The list of OpenXR functions where the OpenXR runtime **may** access the `VkQueue` are:

- xrBeginFrame
- xrEndFrame
- xrAcquireSwapchainImage
- xrReleaseSwapchainImage

The runtime **must** not access the `VkQueue` in any OpenXR function that is not listed above or in an extension definition.

Swapchain Image Layout

When an application acquires a swapchain image by calling xrAcquireSwapchainImage in a session created using XrGraphicsBindingVulkanKHR, the OpenXR runtime **must** guarantee that:

- The image has a memory layout compatible with `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` for color images, or `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL` for depth images.
- The `VkQueue` specified in XrGraphicsBindingVulkanKHR has ownership of the image.

When an application releases a swapchain image by calling xrReleaseSwapchainImage, in a session created using XrGraphicsBindingVulkanKHR, the OpenXR runtime **must** interpret the image as:

- Having a memory layout compatible with `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` for color images, or `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL` for depth images.
- Being owned by the `VkQueue` specified in XrGraphicsBindingVulkanKHR.

The application is responsible for transitioning the swapchain image back to the image layout and queue ownership that the OpenXR runtime requires. If the image is not in a layout compatible with the above specifications the runtime **may** exhibit undefined behaviour.

Swapchain Flag Bits

All XrSwapchainUsageFlags values passed in a session created using XrGraphicsBindingVulkanKHR **must** be interpreted as follows by the runtime, so that the returned swapchain images used by the application may be used as if they were created with at least the specified `VkImageUsageFlagBits` or `VkImageCreateFlagBits` set.

XrSwapchainUsageFlagBits	Corresponding Vulkan flag bit
`XR_SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT`	`VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`
`XR_SWAPCHAIN_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`	`VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`
`XR_SWAPCHAIN_USAGE_UNORDERED_ACCESS_BIT`	`VK_IMAGE_USAGE_STORAGE_BIT`
`XR_SWAPCHAIN_USAGE_TRANSFER_SRC_BIT`	`VK_IMAGE_USAGE_TRANSFER_SRC_BIT`
`XR_SWAPCHAIN_USAGE_TRANSFER_DST_BIT`	`VK_IMAGE_USAGE_TRANSFER_DST_BIT`
`XR_SWAPCHAIN_USAGE_SAMPLED_BIT`	`VK_IMAGE_USAGE_SAMPLED_BIT`
`XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT`	`VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`
`XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_MND` (Added by `XR_MND_swapchain_usage_input_attachment_bit` and only available when that extension is enabled)	`VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN_KHR`
- `XR_TYPE_GRAPHICS_BINDING_VULKAN_KHR`
- `XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR`

New Enums

New Structures

The following structures are provided to supply supporting runtimes the necessary information required to work with the Vulkan API executing on certain operating systems.

The XrGraphicsBindingVulkanKHR structure is defined as:

```
typedef struct XrGraphicsBindingVulkanKHR {
    XrStructureType     type;
    const void*         next;
    VkInstance          instance;
    VkPhysicalDevice    physicalDevice;
    VkDevice            device;
    uint32_t            queueFamilyIndex;
    uint32_t            queueIndex;
} XrGraphicsBindingVulkanKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `instance` is a valid Vulkan `VkInstance`.
- `physicalDevice` is a valid Vulkan `VkPhysicalDevice`.
- `device` is a valid Vulkan `VkDevice`.
- `queueFamilyIndex` is a valid queue family index on `device`.
- `queueIndex` is a valid queue index on `device` to be used for synchronization.

When creating a Vulkan-backed XrSession, the application will provide a pointer to an XrGraphicsBindingVulkanKHR in the `next` chain of the XrSessionCreateInfo.

Valid Usage

- `instance` **must** have enabled a Vulkan API version in the range specified by XrGraphicsBindingVulkanKHR
- `instance` **must** have enabled all the instance extensions specified by xrGetVulkanInstanceExtensionsKHR
- `physicalDevice` `VkPhysicalDevice` **must** match the device specified by xrGetVulkanGraphicsDeviceKHR
- `device` **must** have enabled all the device extensions specified by xrGetVulkanDeviceExtensionsKHR

Valid Usage (Implicit)

- The XR_KHR_vulkan_enable extension **must** be enabled prior to using XrGraphicsBindingVulkanKHR
- type **must** be XR_TYPE_GRAPHICS_BINDING_VULKAN_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- instance **must** be a valid VkInstance value
- physicalDevice **must** be a valid VkPhysicalDevice value
- device **must** be a valid VkDevice value

The XrSwapchainImageVulkanKHR structure is defined as:

```
typedef struct XrSwapchainImageVulkanKHR {
    XrStructureType    type;
    void*              next;
    VkImage            image;
} XrSwapchainImageVulkanKHR;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- image is a valid Vulkan VkImage to use.

If a given session was created with XrGraphicsBindingVulkanKHR, the following conditions **must** apply.

- Calls to xrEnumerateSwapchainImages on an XrSwapchain in that session **must** return an array of XrSwapchainImageVulkanKHR structures.
- Whenever an OpenXR function accepts an XrSwapchainImageBaseHeader pointer as a parameter in that session, the runtime **must** also accept a pointer to an XrSwapchainImageVulkanKHR.

The OpenXR runtime **must** interpret the top-left corner of the swapchain image as the coordinate origin unless specified otherwise by extension functionality.

The OpenXR runtime **must** interpret the swapchain images in a clip space of positive Y pointing down, near Z plane at 0, and far Z plane at 1.

Valid Usage (Implicit)

- The XR_KHR_vulkan_enable extension **must** be enabled prior to using XrSwapchainImageVulkanKHR
- type **must** be XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- image **must** be a valid VkImage value

The XrGraphicsRequirementsVulkanKHR structure is defined as:

```
typedef struct XrGraphicsRequirementsVulkanKHR {
    XrStructureType    type;
    void*              next;
    XrVersion          minApiVersionSupported;
    XrVersion          maxApiVersionSupported;
} XrGraphicsRequirementsVulkanKHR;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- minApiVersionSupported is the minimum Vulkan Instance API version that the runtime supports. Uses XR_MAKE_VERSION on major and minor API version, ignoring any patch version component.
- maxApiVersionSupported is the maximum Vulkan Instance API version that the runtime has been tested on and is known to support. Newer Vulkan Instance API versions might work if they are compatible. Uses XR_MAKE_VERSION on major and minor API version, ignoring any patch version component.

XrGraphicsRequirementsVulkanKHR is populated by xrGetVulkanGraphicsRequirementsKHR with the runtime's Vulkan API version requirements.

Valid Usage (Implicit)

- The XR_KHR_vulkan_enable extension **must** be enabled prior to using XrGraphicsRequirementsVulkanKHR
- type **must** be XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain

New Functions

To query Vulkan API version requirements, call:

```
XrResult xrGetVulkanGraphicsRequirementsKHR(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    XrGraphicsRequirementsVulkanKHR*            graphicsRequirements);
```

Parameter Descriptions

- instance is an XrInstance handle previously created with xrCreateInstance.
- systemId is an XrSystemId handle for the system which will be used to create a session.
- graphicsRequirements is the XrGraphicsRequirementsVulkanKHR output structure.

The xrGetVulkanGraphicsRequirementsKHR function identifies to the application the minimum Vulkan version requirement and the highest known tested Vulkan version. The runtime **must** return XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING (XR_ERROR_VALIDATION_FAILURE **may** be returned due to legacy behavior) on calls to xrCreateSession if xrGetVulkanGraphicsRequirementsKHR has not been called for the same instance and systemId.

Valid Usage (Implicit)

- The XR_KHR_vulkan_enable extension **must** be enabled prior to calling xrGetVulkanGraphicsRequirementsKHR
- instance **must** be a valid XrInstance handle
- graphicsRequirements **must** be a pointer to an XrGraphicsRequirementsVulkanKHR structure

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_SYSTEM_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_FUNCTION_UNSUPPORTED

Some computer systems may have multiple graphics devices, each of which may have independent external display outputs. XR systems that connect to such graphics devices are typically connected to a single device. Applications need to know what graphics device the XR system is connected to so that they can use that graphics device to generate XR images.

To identify what graphics device needs to be used for an instance and system, call:

```
XrResult xrGetVulkanGraphicsDeviceKHR(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    VkInstance                                  vkInstance,
    VkPhysicalDevice*                           vkPhysicalDevice);
```

Parameter Descriptions

- `instance` is an XrInstance handle previously created with xrCreateInstance.
- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `vkInstance` is a valid Vulkan `VkInstance`.
- `vkPhysicalDevice` is a pointer to a `VkPhysicalDevice` value to populate.

xrGetVulkanGraphicsDeviceKHR function identifies to the application what graphics device (Vulkan `VkPhysicalDevice`) needs to be used. xrGetVulkanGraphicsDeviceKHR **must** be called prior to calling xrCreateSession, and the `VkPhysicalDevice` that xrGetVulkanGraphicsDeviceKHR returns should be passed to xrCreateSession in the XrGraphicsBindingVulkanKHR.

Valid Usage (Implicit)

- The XR_KHR_vulkan_enable extension **must** be enabled prior to calling xrGetVulkanGraphicsDeviceKHR
- instance **must** be a valid XrInstance handle
- vkInstance **must** be a valid VkInstance value
- vkPhysicalDevice **must** be a pointer to a VkPhysicalDevice value

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_SYSTEM_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_FUNCTION_UNSUPPORTED

```
XrResult xrGetVulkanInstanceExtensionsKHR(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    uint32_t                                    bufferCapacityInput,
    uint32_t*                                   bufferCountOutput,
    char*                                       buffer);
```

Parameter Descriptions

- `instance` is an XrInstance handle previously created with xrCreateInstance.
- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `bufferCapacityInput` is the capacity of the `buffer`, or 0 to indicate a request to retrieve the required capacity.
- `bufferCountOutput` is a pointer to the count of characters written (including terminating `\0`), or a pointer to the required capacity in the case that `bufferCapacityInput` is 0.
- `buffer` is a pointer to an array of characters, but **can** be `NULL` if `bufferCapacityInput` is 0. The format of the output is a single space (ASCII `0x20`) delimited string of extension names.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required `buffer` size.

Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable` extension **must** be enabled prior to calling xrGetVulkanInstanceExtensionsKHR
- `instance` **must** be a valid XrInstance handle
- `bufferCountOutput` **must** be a pointer to a `uint32_t` value
- If `bufferCapacityInput` is not `0`, `buffer` **must** be a pointer to an array of `bufferCapacityInput` char values

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_SYSTEM_INVALID`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_SIZE_INSUFFICIENT`
- `XR_ERROR_FUNCTION_UNSUPPORTED`

```
XrResult xrGetVulkanDeviceExtensionsKHR(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    uint32_t                                    bufferCapacityInput,
    uint32_t*                                   bufferCountOutput,
    char*                                       buffer);
```

Parameter Descriptions

- `instance` is an XrInstance handle previously created with xrCreateInstance.
- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `bufferCapacityInput` is the capacity of the `buffer`, or 0 to indicate a request to retrieve the required capacity.
- `bufferCountOutput` is a pointer to the count of characters written (including terminating `\0`), or a pointer to the required capacity in the case that `bufferCapacityInput` is 0.
- `buffer` is a pointer to an array of characters, but **can** be `NULL` if `bufferCapacityInput` is 0. The format of the output is a single space (ASCII `0x20`) delimited string of extension names.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required `buffer` size.

Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable` extension **must** be enabled prior to calling xrGetVulkanDeviceExtensionsKHR
- `instance` **must** be a valid XrInstance handle
- `bufferCountOutput` **must** be a pointer to a `uint32_t` value
- If `bufferCapacityInput` is not `0`, `buffer` **must** be a pointer to an array of `bufferCapacityInput` char values

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_SYSTEM_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_SIZE_INSUFFICIENT
- XR_ERROR_FUNCTION_UNSUPPORTED

Issues

Version History

- Revision 1, 2018-05-07 (Mark Young)
  - Initial draft
- Revision 2, 2018-06-21 (Bryce Hutchings)
  - Replace `session` parameter with `instance` and `systemId` parameters.
  - Move `xrGetVulkanDeviceExtensionsKHR`, `xrGetVulkanInstanceExtensionsKHR` and `xrGetVulkanGraphicsDeviceKHR` functions into this extension
  - Add new `XrGraphicsRequirementsVulkanKHR` function.
- Revision 3, 2018-11-15 (Paul Pedriana)
  - Specified the swapchain texture coordinate origin.
- Revision 4, 2018-11-16 (Minmin Gong)
  - Specified Y direction and Z range in clip space
- Revision 5, 2019-01-24 (Robert Menzel)
  - Description updated
- Revision 6, 2019-01-25 (Andres Rodriguez)
  - Reword sections of the spec to shift requirements on to the runtime instead of the app
- Revision 7, 2020-08-06 (Bryce Hutchings)
  - Added new `XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING` error code

- Revision 8, 2021-01-21 (Ryan Pavlik)
  - Document mapping for XrSwapchainUsageFlags

12.20. XR_KHR_vulkan_enable2

Name String

XR_KHR_vulkan_enable2

Extension Type

Instance extension

Registered Extension Number

91

Revision

2

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2020-05-04

IP Status

No known IP claims.

Contributors

Mark Young, LunarG
Paul Pedriana, Oculus
Ed Hutchins, Oculus
Andres Rodriguez, Valve
Dan Ginsburg, Valve
Bryce Hutchings, Microsoft
Minmin Gong, Microsoft
Robert Menzel, NVIDIA
Paulo F. Gomes, Samsung Electronics

12.20.1. Overview

This extension enables the use of the Vulkan graphics API in an OpenXR runtime. Without this extension, the OpenXR runtime may not be able to use any Vulkan swapchain images.

This extension provides the mechanisms necessary for an application to generate a valid XrGraphicsBindingVulkan2KHR structure in order to create a Vulkan-based XrSession.

This extension also provides mechanisms for the application to interact with images acquired by calling xrEnumerateSwapchainImages.

In order to expose the structures, types, and functions of this extension, you **must** define XR_USE_GRAPHICS_API_VULKAN before including the OpenXR platform header `openxr_platform.h`, in all portions of your library or application that include it.

> *Note*
>
> This extension is intended as an alternative to `XR_KHR_vulkan_enable`, and does not depend on it.

12.20.2. Initialization

When operating in Vulkan mode, the OpenXR runtime and the application will share the Vulkan queue described in the XrGraphicsBindingVulkan2KHR structure. This section of the document describes the mechanisms this extension exposes to ensure the shared Vulkan queue is compatible with the runtime and the application's requirements.

Vulkan Version Requirements

First, a compatible Vulkan version **must** be agreed upon. To query the runtime's Vulkan API version requirements an application will call:

```
XrResult xrGetVulkanGraphicsRequirements2KHR(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    XrGraphicsRequirementsVulkanKHR*            graphicsRequirements);
```

The xrGetVulkanGraphicsRequirements2KHR function identifies to the application the runtime's minimum Vulkan version requirement and the highest known tested Vulkan version. xrGetVulkanGraphicsRequirements2KHR **must** be called prior to calling xrCreateSession. The runtime **must** return `XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING` on calls to xrCreateSession if xrGetVulkanGraphicsRequirements2KHR has not been called for the same `instance` and `systemId`.

Parameter Descriptions

- `instance` is an XrInstance handle previously created with xrCreateInstance.
- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `graphicsRequirements` is the XrGraphicsRequirementsVulkan2KHR output structure.

Valid Usage (Implicit)

- The XR_KHR_vulkan_enable2 extension **must** be enabled prior to calling xrGetVulkanGraphicsRequirements2KHR
- instance **must** be a valid XrInstance handle
- graphicsRequirements **must** be a pointer to an XrGraphicsRequirementsVulkanKHR structure

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_SYSTEM_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_FUNCTION_UNSUPPORTED

The XrGraphicsRequirementsVulkan2KHR structure populated by xrGetVulkanGraphicsRequirements2KHR is defined as:

```
// XrGraphicsRequirementsVulkan2KHR is an alias for XrGraphicsRequirementsVulkanKHR
typedef struct XrGraphicsRequirementsVulkanKHR {
    XrStructureType    type;
    void*              next;
    XrVersion          minApiVersionSupported;
    XrVersion          maxApiVersionSupported;
} XrGraphicsRequirementsVulkanKHR;

typedef XrGraphicsRequirementsVulkanKHR XrGraphicsRequirementsVulkan2KHR;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- minApiVersionSupported is the minimum version of Vulkan that the runtime supports. Uses XR_MAKE_VERSION on major and minor API version, ignoring any patch version component.
- maxApiVersionSupported is the maximum version of Vulkan that the runtime has been tested on and is known to support. Newer Vulkan versions might work if they are compatible. Uses XR_MAKE_VERSION on major and minor API version, ignoring any patch version component.

Valid Usage (Implicit)

- The XR_KHR_vulkan_enable2 extension **must** be enabled prior to using XrGraphicsRequirementsVulkan2KHR
- **Note:** XrGraphicsRequirementsVulkan2KHR is an alias for XrGraphicsRequirementsVulkanKHR, so implicit valid usage for XrGraphicsRequirementsVulkanKHR has been replaced below.
- type **must** be XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain

Vulkan Instance Creation

Second, a compatible VkInstance **must** be created. The xrCreateVulkanInstanceKHR entry point is a wrapper around vkCreateInstance intended for this purpose. When called, the runtime **must** aggregate the requirements specified by the application with its own requirements and forward the VkInstance creation request to the vkCreateInstance function pointer returned by pfnGetInstanceProcAddr.

```
XrResult xrCreateVulkanInstanceKHR(
    XrInstance                                  instance,
    const XrVulkanInstanceCreateInfoKHR*        createInfo,
    VkInstance*                                 vulkanInstance,
    VkResult*                                   vulkanResult);
```

Parameter Descriptions

- `instance` is an XrInstance handle previously created with xrCreateInstance.
- `createInfo` extensible input struct of type `XrCreateVulkanInstanceCreateInfoKHR`
- `vulkanInstance` points to a `VkInstance` handle to populate with the new Vulkan instance.
- `vulkanResult` points to a `VkResult` to populate with the result of the `vkCreateInstance` operation as returned by `pfnGetInstanceProcAddr`.

Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable2` extension **must** be enabled prior to calling xrCreateVulkanInstanceKHR
- `instance` **must** be a valid XrInstance handle
- `createInfo` **must** be a pointer to a valid XrVulkanInstanceCreateInfoKHR structure
- `vulkanInstance` **must** be a pointer to a `VkInstance` value
- `vulkanResult` **must** be a pointer to a `VkResult` value

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_SYSTEM_INVALID`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_FUNCTION_UNSUPPORTED`

The XrVulkanInstanceCreateInfoKHR structure contains the input parameters to xrCreateVulkanInstanceKHR.

```
typedef struct XrVulkanInstanceCreateInfoKHR {
    XrStructureType                    type;
    const void*                        next;
    XrSystemId                         systemId;
    XrVulkanInstanceCreateFlagsKHR     createFlags;
    PFN_vkGetInstanceProcAddr          pfnGetInstanceProcAddr;
    const VkInstanceCreateInfo*        vulkanCreateInfo;
    const VkAllocationCallbacks*       vulkanAllocator;
} XrVulkanInstanceCreateInfoKHR;
```

Member Descriptions

- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `pfnGetInstanceProcAddr` is a function pointer to `vkGetInstanceProcAddr` or a compatible entry point.
- `vulkanCreateInfo` is the `VkInstanceCreateInfo` as specified by Vulkan.
- `vulkanAllocator` is the `VkAllocationCallbacks` as specified by Vulkan.

Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable2` extension **must** be enabled prior to using XrVulkanInstanceCreateInfoKHR
- `type` **must** be `XR_TYPE_VULKAN_INSTANCE_CREATE_INFO_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `createFlags` **must** be `0`
- `pfnGetInstanceProcAddr` **must** be a valid `PFN_vkGetInstanceProcAddr` value
- `vulkanCreateInfo` **must** be a pointer to a valid `VkInstanceCreateInfo` value
- If `vulkanAllocator` is not `NULL`, `vulkanAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` value

```
typedef XrFlags64 XrVulkanInstanceCreateFlagsKHR;
```

Physical Device Selection

Third, a `VkPhysicalDevice` **must** be chosen. Some computer systems may have multiple graphics devices, each of which may have independent external display outputs. The runtime **must** report a

VkPhysicalDevice that is compatible with the OpenXR implementation when xrGetVulkanGraphicsDevice2KHR is invoked. The application will use this VkPhysicalDevice to interact with the OpenXR runtime.

```
XrResult xrGetVulkanGraphicsDevice2KHR(
    XrInstance                                  instance,
    const XrVulkanGraphicsDeviceGetInfoKHR*     getInfo,
    VkPhysicalDevice*                           vulkanPhysicalDevice);
```

Parameter Descriptions

- instance is an XrInstance handle previously created with xrCreateInstance.
- getInfo extensible input struct of type XrVulkanGraphicsDeviceGetInfoKHR
- vulkanPhysicalDevice is a pointer to a VkPhysicalDevice handle to populate.

Valid Usage (Implicit)

- The XR_KHR_vulkan_enable2 extension **must** be enabled prior to calling xrGetVulkanGraphicsDevice2KHR
- instance **must** be a valid XrInstance handle
- getInfo **must** be a pointer to a valid XrVulkanGraphicsDeviceGetInfoKHR structure
- vulkanPhysicalDevice **must** be a pointer to a VkPhysicalDevice value

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_SYSTEM_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_FUNCTION_UNSUPPORTED

The XrVulkanGraphicsDeviceGetInfoKHR structure contains the input parameters to xrCreateVulkanInstanceKHR.

```
typedef struct XrVulkanGraphicsDeviceGetInfoKHR {
    XrStructureType    type;
    const void*        next;
    XrSystemId         systemId;
    VkInstance         vulkanInstance;
} XrVulkanGraphicsDeviceGetInfoKHR;
```

Member Descriptions

- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `vulkanInstance` is a valid Vulkan `VkInstance`.

Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable2` extension **must** be enabled prior to using XrVulkanGraphicsDeviceGetInfoKHR
- `type` **must** be `XR_TYPE_VULKAN_GRAPHICS_DEVICE_GET_INFO_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `vulkanInstance` **must** be a valid `VkInstance` value

Vulkan Device Creation

Fourth, a compatible `VkDevice` **must** be created. The xrCreateVulkanDeviceKHR entry point is a wrapper around vkCreateDevice intended for this purpose. When called, the runtime **must** aggregate the requirements specified by the application with its own requirements and forward the `VkDevice` creation request to the `vkCreateDevice` function pointer returned by `pfnGetInstanceProcAddr`.

```
XrResult xrCreateVulkanDeviceKHR(
    XrInstance                                  instance,
    const XrVulkanDeviceCreateInfoKHR*          createInfo,
    VkDevice*                                   vulkanDevice,
    VkResult*                                   vulkanResult);
```

Parameter Descriptions

- instance is an XrInstance handle previously created with xrCreateInstance.
- createInfo extensible input struct of type XrCreateVulkanDeviceCreateInfoKHR
- vulkanDevice points to a VkDevice handle to populate with the new Vulkan device.
- vulkanResult points to a VkResult to populate with the result of the vkCreateDevice operation as returned by pfnGetInstanceProcAddr.

Valid Usage (Implicit)

- The XR_KHR_vulkan_enable2 extension **must** be enabled prior to calling xrCreateVulkanDeviceKHR
- instance **must** be a valid XrInstance handle
- createInfo **must** be a pointer to a valid XrVulkanDeviceCreateInfoKHR structure
- vulkanDevice **must** be a pointer to a VkDevice value
- vulkanResult **must** be a pointer to a VkResult value

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_SYSTEM_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_FUNCTION_UNSUPPORTED

The XrVulkanDeviceCreateInfoKHR structure contains the input parameters to xrCreateVulkanDeviceKHR.

```
typedef struct XrVulkanDeviceCreateInfoKHR {
    XrStructureType                  type;
    const void*                      next;
    XrSystemId                       systemId;
    XrVulkanDeviceCreateFlagsKHR     createFlags;
    PFN_vkGetInstanceProcAddr        pfnGetInstanceProcAddr;
    VkPhysicalDevice                 vulkanPhysicalDevice;
    const VkDeviceCreateInfo*        vulkanCreateInfo;
    const VkAllocationCallbacks*     vulkanAllocator;
} XrVulkanDeviceCreateInfoKHR;
```

Member Descriptions

- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `pfnGetInstanceProcAddr` is a function pointer to `vkGetInstanceProcAddr` or a compatible entry point.
- `vulkanPhysicalDevice` **must** match xrGetVulkanGraphicsDeviceKHR.
- `vulkanCreateInfo` is the `VkDeviceCreateInfo` as specified by Vulkan.
- `vulkanAllocator` is the `VkAllocationCallbacks` as specified by Vulkan.

Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable2` extension **must** be enabled prior to using XrVulkanDeviceCreateInfoKHR
- `type` **must** be `XR_TYPE_VULKAN_DEVICE_CREATE_INFO_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `createFlags` **must** be `0`
- `pfnGetInstanceProcAddr` **must** be a valid `PFN_vkGetInstanceProcAddr` value
- `vulkanPhysicalDevice` **must** be a valid `VkPhysicalDevice` value
- `vulkanCreateInfo` **must** be a pointer to a valid `VkDeviceCreateInfo` value
- If `vulkanAllocator` is not `NULL`, `vulkanAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` value

```
typedef XrFlags64 XrVulkanDeviceCreateFlagsKHR;
```

If the `vulkanPhysicalDevice` parameter does not match the output of xrGetVulkanGraphicsDeviceKHR, then the runtime **must** return `XR_ERROR_HANDLE_INVALID`.

Queue Selection

Last, the application selects a `VkQueue` from the `VkDevice` that has the `VK_QUEUE_GRAPHICS_BIT` set.

> *Note*
>
> The runtime may schedule work on the `VkQueue` specified in the binding, or it may schedule work on any hardware queue in a foreign logical device.

Vulkan Graphics Binding

When creating a Vulkan-backed XrSession, the application will chain a pointer to an XrGraphicsBindingVulkan2KHR to the XrSessionCreateInfo parameter of xrCreateSession. With the data collected in the previous sections, the application now has all the necessary information to populate an XrGraphicsBindingVulkan2KHR structure for session creation.

```
// XrGraphicsBindingVulkan2KHR is an alias for XrGraphicsBindingVulkanKHR
typedef struct XrGraphicsBindingVulkanKHR {
    XrStructureType     type;
    const void*         next;
    VkInstance          instance;
    VkPhysicalDevice    physicalDevice;
    VkDevice            device;
    uint32_t            queueFamilyIndex;
    uint32_t            queueIndex;
} XrGraphicsBindingVulkanKHR;

typedef XrGraphicsBindingVulkanKHR XrGraphicsBindingVulkan2KHR;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- instance is a valid Vulkan VkInstance.
- physicalDevice is a valid Vulkan VkPhysicalDevice.
- device is a valid Vulkan VkDevice.
- queueFamilyIndex is a valid queue family index on device.
- queueIndex is a valid queue index on device to be used for synchronization.

Valid Usage

- instance **must** have enabled a Vulkan API version in the range specified by xrGetVulkanGraphicsRequirements2KHR
- instance **must** have been created using xrCreateVulkanInstanceKHR
- physicalDevice VkPhysicalDevice **must** match the device specified by xrGetVulkanGraphicsDevice2KHR
- device **must** have been created using xrCreateVulkanDeviceKHR

Valid Usage (Implicit)

- The XR_KHR_vulkan_enable2 extension **must** be enabled prior to using XrGraphicsBindingVulkan2KHR
- **Note:** XrGraphicsBindingVulkan2KHR is an alias for XrGraphicsBindingVulkanKHR, so implicit valid usage for XrGraphicsBindingVulkanKHR has been replicated below.
- type **must** be XR_TYPE_GRAPHICS_BINDING_VULKAN_KHR
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- instance **must** be a valid VkInstance value
- physicalDevice **must** be a valid VkPhysicalDevice value
- device **must** be a valid VkDevice value

Populating a XrGraphicsBindingVulkan2KHR structure with a member that does not meet the requirements outlined by this extension **may** result in undefined behaviour by the OpenXR runtime.

The requirements outlined in this extension only apply to the VkInstance, VkDevice, VkPhysicalDevice and VkQueue objects which the application wishes to associate with an XrGraphicsBindingVulkan2KHR.

12.20.3. Concurrency

Vulkan requires that concurrent access to a VkQueue from multiple threads be externally synchronized. Therefore, OpenXR functions that may access the VkQueue specified in the XrGraphicsBindingVulkan2KHR **must** also be externally synchronized by the OpenXR application.

The list of OpenXR functions where the OpenXR runtime **may** access the VkQueue are:

- xrBeginFrame
- xrEndFrame
- xrAcquireSwapchainImage
- xrReleaseSwapchainImage

The runtime **must** not access the VkQueue in any OpenXR function that is not listed above or in an extension definition.

Failure by the application to synchronize access to VkQueue **may** result in undefined behaviour in the OpenXR runtime.

12.20.4. Swapchain Interactions

Swapchain Images

When an application interacts with XrSwapchainImageBaseHeader structures in a Vulkan-backed XrSession, the application can interpret these to be XrSwapchainImageVulkan2KHR structures. These are defined as:

```
// XrSwapchainImageVulkan2KHR is an alias for XrSwapchainImageVulkanKHR
typedef struct XrSwapchainImageVulkanKHR {
    XrStructureType    type;
    void*              next;
    VkImage            image;
} XrSwapchainImageVulkanKHR;

typedef XrSwapchainImageVulkanKHR XrSwapchainImageVulkan2KHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `image` is a valid Vulkan `VkImage` to use.

If a given session was created with XrGraphicsBindingVulkan2KHR, the following conditions **must** apply.

- Calls to xrEnumerateSwapchainImages on an XrSwapchain in that session **must** return an array of XrSwapchainImageVulkan2KHR structures.
- Whenever an OpenXR function accepts an XrSwapchainImageBaseHeader pointer as a parameter in that session, the runtime **must** also accept a pointer to an XrSwapchainImageVulkan2KHR.

The OpenXR runtime **must** interpret the top-left corner of the swapchain image as the coordinate origin unless specified otherwise by extension functionality.

The OpenXR runtime **must** interpret the swapchain images in a clip space of positive Y pointing down, near Z plane at 0, and far Z plane at 1.

Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable2` extension **must** be enabled prior to using XrSwapchainImageVulkan2KHR
- **Note:** XrSwapchainImageVulkan2KHR is an alias for XrSwapchainImageVulkanKHR, so implicit valid usage for XrSwapchainImageVulkanKHR has been replicated below.
- `type` **must** be `XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `image` **must** be a valid `VkImage` value

Swapchain Image Layout

When an application acquires a swapchain image by calling xrAcquireSwapchainImage in a session created using XrGraphicsBindingVulkan2KHR, the OpenXR runtime **must** guarantee that:

- The image has a memory layout compatible with `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` for color images, or `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL` for depth images.
- The `VkQueue` specified in XrGraphicsBindingVulkan2KHR has ownership of the image.

When an application releases a swapchain image by calling xrReleaseSwapchainImage, in a session created using XrGraphicsBindingVulkan2KHR, the OpenXR runtime **must** interpret the image as:

- Having a memory layout compatible with VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL for color images, or VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL for depth images.
- Being owned by the VkQueue specified in XrGraphicsBindingVulkan2KHR.
- Being referenced by command buffers submitted to the VkQueue specified in XrGraphicsBindingVulkan2KHR which have not yet completed execution.

The application is responsible for transitioning the swapchain image back to the image layout and queue ownership that the OpenXR runtime requires. If the image is not in a layout compatible with the above specifications the runtime **may** exhibit undefined behaviour.

Swapchain Flag Bits

All XrSwapchainUsageFlags values passed in a session created using XrGraphicsBindingVulkan2KHR **must** be interpreted as follows by the runtime, so that the returned swapchain images used by the application may be used as if they were created with at least the specified VkImageUsageFlagBits or VkImageCreateFlagBits set.

XrSwapchainUsageFlagBits	Corresponding Vulkan flag bit
XR_SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT	VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
XR_SWAPCHAIN_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT	VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
XR_SWAPCHAIN_USAGE_UNORDERED_ACCESS_BIT	VK_IMAGE_USAGE_STORAGE_BIT
XR_SWAPCHAIN_USAGE_TRANSFER_SRC_BIT	VK_IMAGE_USAGE_TRANSFER_SRC_BIT
XR_SWAPCHAIN_USAGE_TRANSFER_DST_BIT	VK_IMAGE_USAGE_TRANSFER_DST_BIT
XR_SWAPCHAIN_USAGE_SAMPLED_BIT	VK_IMAGE_USAGE_SAMPLED_BIT
XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT	VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT
XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_MND (Added by XR_MND_swapchain_usage_input_attachment_bit and only available when that extension is enabled)	VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT

12.20.5. Appendix

Questions

1. Should the xrCreateVulkanDeviceKHR and xrCreateVulkanInstanceKHR functions have an output parameter that returns the combined list of parameters used to create the Vulkan device/instance?
   - No. If the application is interested in capturing this data it can set the pfnGetInstanceProcAddr parameter to a local callback that captures the relevant information.

Quick Reference

New Enum Constants

XrStructureType enumeration is extended with:

- XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN2_KHR (alias of XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN_KHR)
- XR_TYPE_GRAPHICS_BINDING_VULKAN2_KHR (alias of XR_TYPE_GRAPHICS_BINDING_VULKAN_KHR)
- XR_TYPE_SWAPCHAIN_IMAGE_VULKAN2_KHR (alias of XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR)

New Structures

- XrVulkanInstanceCreateInfoKHR
- XrVulkanDeviceCreateInfoKHR
- XrVulkanGraphicsDeviceGetInfoKHR
- XrGraphicsBindingVulkan2KHR (alias of XrGraphicsBindingVulkanKHR)
- XrSwapchainImageVulkan2KHR (alias of XrSwapchainImageVulkanKHR)
- XrGraphicsRequirementsVulkan2KHR (alias of XrGraphicsRequirementsVulkanKHR)

New Functions

- xrCreateVulkanInstanceKHR
- xrCreateVulkanDeviceKHR
- xrGetVulkanGraphicsDevice2KHR
- xrGetVulkanGraphicsRequirements2KHR

Version History

- Revision 1, 2020-05-04 (Andres Rodriguez)
  - Initial draft
- Revision 2, 2021-01-21 (Ryan Pavlik)
  - Document mapping for XrSwapchainUsageFlags

12.21. XR_KHR_vulkan_swapchain_format_list

Name String

XR_KHR_vulkan_swapchain_format_list

Extension Type

Instance extension

Registered Extension Number

15

Revision

4

Extension and Version Dependencies

- Requires OpenXR 1.0
- Requires XR_KHR_vulkan_enable

Last Modified Date

2020-01-01

IP Status

No known IP claims.

Contributors

Paul Pedriana, Oculus
Dan Ginsburg, Valve

Overview

Vulkan has the `VK_KHR_image_format_list` extension which allows applications to tell the `vkCreateImage` function which formats the application intends to use when `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` is specified. This OpenXR extension exposes that Vulkan extension to OpenXR applications. In the same way that a Vulkan-based application can pass a `VkImageFormatListCreateInfo` struct to the `vkCreateImage` function, an OpenXR application can pass an identically configured XrVulkanSwapchainFormatListCreateInfoKHR structure to xrCreateSwapchain.

Applications using this extension to specify more than one swapchain format must create OpenXR swapchains with the `XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT` bit set.

Runtimes implementing this extension **must** support the `XR_KHR_vulkan_enable` or the `XR_KHR_vulkan_enable2` extension. When `XR_KHR_vulkan_enable` is used, the runtime **must** add `VK_KHR_image_format_list` to the list of extensions enabled in xrCreateVulkanDeviceKHR.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

```
XR_TYPE_VULKAN_SWAPCHAIN_FORMAT_LIST_CREATE_INFO_KHR
```

New Enums

New Structures

```
typedef struct XrVulkanSwapchainFormatListCreateInfoKHR {
    XrStructureType    type;
    const void*        next;
    uint32_t           viewFormatCount;
    const VkFormat*    viewFormats;
} XrVulkanSwapchainFormatListCreateInfoKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `viewFormatCount` is the number of view formats passed in `viewFormats`.
- `viewFormats` is an array of `VkFormat`.

Valid Usage (Implicit)

- The `XR_KHR_vulkan_swapchain_format_list` extension **must** be enabled prior to using XrVulkanSwapchainFormatListCreateInfoKHR
- `type` **must** be `XR_TYPE_VULKAN_SWAPCHAIN_FORMAT_LIST_CREATE_INFO_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- If `viewFormatCount` is not `0`, `viewFormats` **must** be a pointer to an array of `viewFormatCount` valid `VkFormat` values

New Functions

Issues

Version History

- Revision 1, 2017-09-13 (Paul Pedriana)
  - Initial proposal.
- Revision 2, 2018-06-21 (Bryce Hutchings)

- Update reference of XR_KHR_vulkan_extension_requirements to XR_KHR_vulkan_enable
- Revision 3, 2020-01-01 (Andres Rodriguez)
  - Update for XR_KHR_vulkan_enable2
- Revision 4, 2021-01-21 (Ryan Pavlik)
  - Fix reference to the mutable-format bit in Vulkan.

12.22. XR_KHR_win32_convert_performance_counter_time

Name String

XR_KHR_win32_convert_performance_counter_time

Extension Type

Instance extension

Registered Extension Number

36

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2019-01-24

IP Status

No known IP claims.

Contributors

Paul Pedriana, Oculus
Bryce Hutchings, Microsoft

Overview

This extension provides two functions for converting between the Windows performance counter (QPC) time stamps and XrTime. The xrConvertWin32PerformanceCounterToTimeKHR function converts from Windows performance counter time stamps to XrTime, while the xrConvertTimeToWin32PerformanceCounterKHR function converts XrTime to Windows performance counter time stamps. The primary use case for this functionality is to be able to synchronize events between the local system and the OpenXR system.

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

New Functions

To convert from a Windows performance counter time stamp to `XrTime`, call:

```
XrResult xrConvertWin32PerformanceCounterToTimeKHR(
    XrInstance                                  instance,
    const LARGE_INTEGER*                        performanceCounter,
    XrTime*                                     time);
```

Parameter Descriptions

- `instance` is an XrInstance handle previously created with xrCreateInstance.
- `performanceCounter` is a time returned by `QueryPerformanceCounter`.
- `time` is the resulting `XrTime` that is equivalent to the `performanceCounter`.

The xrConvertWin32PerformanceCounterToTimeKHR function converts a time stamp obtained by the `QueryPerformanceCounter` Windows function to the equivalent `XrTime`.

If the output `time` cannot represent the input `performanceCounter`, the runtime **must** return `XR_ERROR_TIME_INVALID`.

Valid Usage (Implicit)

- The `XR_KHR_win32_convert_performance_counter_time` extension **must** be enabled prior to calling xrConvertWin32PerformanceCounterToTimeKHR
- `instance` **must** be a valid XrInstance handle
- `performanceCounter` **must** be a pointer to a valid `LARGE_INTEGER` value
- `time` **must** be a pointer to an `XrTime` value

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_TIME_INVALID
- XR_ERROR_FUNCTION_UNSUPPORTED

To convert from XrTime to a Windows performance counter time stamp, call:

```
XrResult xrConvertTimeToWin32PerformanceCounterKHR(
    XrInstance                                  instance,
    XrTime                                      time,
    LARGE_INTEGER*                              performanceCounter);
```

Parameter Descriptions

- instance is an XrInstance handle previously created with xrCreateInstance.
- time is an XrTime.
- performanceCounter is the resulting Windows performance counter time stamp that is equivalent to the time.

The xrConvertTimeToWin32PerformanceCounterKHR function converts an XrTime to time as if generated by the QueryPerformanceCounter Windows function.

If the output performanceCounter cannot represent the input time, the runtime **must** return XR_ERROR_TIME_INVALID.

Valid Usage (Implicit)

- The `XR_KHR_win32_convert_performance_counter_time` extension **must** be enabled prior to calling xrConvertTimeToWin32PerformanceCounterKHR
- `instance` **must** be a valid XrInstance handle
- `performanceCounter` **must** be a pointer to a `LARGE_INTEGER` value

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_TIME_INVALID`
- `XR_ERROR_FUNCTION_UNSUPPORTED`

Issues

Version History

- Revision 1, 2019-01-24 (Paul Pedriana)
  - Initial draft

12.23. XR_EXT_conformance_automation

Name String

`XR_EXT_conformance_automation`

Extension Type

Instance extension

Registered Extension Number

48

Revision

3

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2021-04-14

IP Status

No known IP claims.

Contributors

Lachlan Ford, Microsoft
Ryan Pavlik, Collabora

Overview

The XR_EXT_conformance_automation allows conformance test and runtime developers to provide hints to the underlying runtime as to what input the test is expecting. This enables runtime authors to automate the testing of their runtime conformance. This is useful for achieving rapidly iterative runtime development whilst maintaining conformance for runtime releases.

This extension provides the following capabilities:

- The ability to toggle the active state of an input device.
- The ability to set the state of an input device button or other input component.
- The ability to set the location of the input device.

Applications **may** call these functions at any time. The runtime **must** do its best to honor the request of applications calling these functions, however it does not guarantee that any state change will be reflected immediately, at all, or with the exact value that was requested. Applications are thus advised to wait for the state change to be observable and to not assume that the value they requested will be the value observed. If any of the functions of this extension are called, control over input **must** be removed from the physical hardware of the system.

> **Warning**
>
> This extension is **not** intended for use by non-conformance-test applications. A runtime **may** require a runtime-specified configuration such as a "developer mode" to be enabled before reporting support for this extension or providing a non-stub implementation of it.
>
> **Do not** use this functionality in a non-conformance-test application!

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

New Functions

```
XrResult xrSetInputDeviceActiveEXT(
    XrSession                                   session,
    XrPath                                      interactionProfile,
    XrPath                                      topLevelPath,
    XrBool32                                    isActive);
```

Parameter Descriptions

- `session` is the XrSession to set the input device state in.
- `interactionProfile` is the path representing the interaction profile of the input device (e.g. */interaction_profiles/khr/simple_controller*).
- `topLevelPath` is the path representing the input device (e.g. */user/hand/left*).
- `isActive` is the requested activation state of the input device.

Valid Usage

- `session` **must** be a valid session handle.
- `topLevelPath` **must** be a valid top level path.

Valid Usage (Implicit)

- The `XR_EXT_conformance_automation` extension **must** be enabled prior to calling xrSetInputDeviceActiveEXT
- `session` **must** be a valid XrSession handle

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_SESSION_LOSS_PENDING
- XR_ERROR_PATH_INVALID
- XR_ERROR_PATH_UNSUPPORTED

```
XrResult xrSetInputDeviceStateBoolEXT(
    XrSession                                   session,
    XrPath                                      topLevelPath,
    XrPath                                      inputSourcePath,
    XrBool32                                    state);
```

Parameter Descriptions

- session is the XrSession to set the input device state in.
- topLevelPath is the path representing the input device (e.g. */user/hand/left*).
- inputSourcePath is the full path of the input component for which we wish to set the state for (e.g. */user/hand/left/input/select/click*).
- state is the requested boolean state of the input device.

Valid Usage

- session **must** be a valid session handle.
- topLevelPath **must** be a valid top level path.
- inputSourcePath **must** be a valid input source path.

Valid Usage (Implicit)

- The `XR_EXT_conformance_automation` extension **must** be enabled prior to calling xrSetInputDeviceStateBoolEXT
- `session` **must** be a valid XrSession handle

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_SESSION_LOSS_PENDING`
- `XR_ERROR_PATH_INVALID`
- `XR_ERROR_PATH_UNSUPPORTED`

```
XrResult xrSetInputDeviceStateFloatEXT(
    XrSession                                   session,
    XrPath                                      topLevelPath,
    XrPath                                      inputSourcePath,
    float                                       state);
```

Parameter Descriptions

- `session` is the XrSession to set the input device state in.
- `topLevelPath` is the path representing the input device (e.g. */user/hand/left*).
- `inputSourcePath` is the full path of the input component for which we wish to set the state for (e.g. */user/hand/left/input/trigger/value*).
- `state` is the requested float state of the input device.

Valid Usage

- session **must** be a valid session handle.
- topLevelPath **must** be a valid top level path.
- inputSourcePath **must** be a valid input source path.

Valid Usage (Implicit)

- The XR_EXT_conformance_automation extension **must** be enabled prior to calling xrSetInputDeviceStateFloatEXT
- session **must** be a valid XrSession handle

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_SESSION_LOSS_PENDING
- XR_ERROR_PATH_INVALID
- XR_ERROR_PATH_UNSUPPORTED

```
XrResult xrSetInputDeviceStateVector2fEXT(
    XrSession                                   session,
    XrPath                                      topLevelPath,
    XrPath                                      inputSourcePath,
    XrVector2f                                  state);
```

Parameter Descriptions

- `session` is the XrSession to set the input device state in.
- `topLevelPath` is the path representing the input device (e.g. */user/hand/left*).
- `inputSourcePath` is the full path of the input component for which we wish to set the state for (e.g. */user/hand/left/input/thumbstick*).
- `state` is the requested two-dimensional state of the input device.

Valid Usage

- `session` **must** be a valid session handle.
- `topLevelPath` **must** be a valid top level path.
- `inputSourcePath` **must** be a valid input source path.

Valid Usage (Implicit)

- The `XR_EXT_conformance_automation` extension **must** be enabled prior to calling xrSetInputDeviceStateVector2fEXT
- `session` **must** be a valid XrSession handle

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_SESSION_LOSS_PENDING`
- `XR_ERROR_PATH_INVALID`
- `XR_ERROR_PATH_UNSUPPORTED`

```
XrResult xrSetInputDeviceLocationEXT(
    XrSession                                   session,
    XrPath                                      topLevelPath,
    XrPath                                      inputSourcePath,
    XrSpace                                     space,
    XrPosef                                     pose);
```

Parameter Descriptions

- `session` is the XrSession to set the input device state in.
- `topLevelPath` is the path representing the input device (e.g. */user/hand/left*).
- `inputSourcePath` is the full path of the input component for which we wish to set the pose for (e.g. */user/hand/left/input/grip/pose*).
- `pose` is the requested pose state of the input device.

Valid Usage

- `session` **must** be a valid session handle.
- `topLevelPath` **must** be a valid top level path.
- `inputSourcePath` **must** be a valid input source path.

Valid Usage (Implicit)

- The `XR_EXT_conformance_automation` extension **must** be enabled prior to calling xrSetInputDeviceLocationEXT
- `session` **must** be a valid XrSession handle
- `space` **must** be a valid XrSpace handle
- `space` **must** have been created, allocated, or retrieved from `session`

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_SESSION_LOSS_PENDING
- XR_ERROR_PATH_INVALID
- XR_ERROR_PATH_UNSUPPORTED
- XR_ERROR_POSE_INVALID

New Function Pointers

Issues

None

Version History

- Revision 1, 2019-10-01 (Lachlan Ford)
  - Initial draft
- Revision 2, 2021-03-04 (Ryan Pavlik)
  - Correct errors in function parameter documentation.
- Revision 3, 2021-04-14 (Ryan Pavlik, Collabora)
  - Fix missing error code

12.24. XR_EXT_debug_utils

Name String

XR_EXT_debug_utils

Extension Type

Instance extension

Registered Extension Number

20

Revision

4

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2021-04-14

IP Status

No known IP claims.

Contributors

Mark Young, LunarG
Karl Schultz, LunarG
Ryan Pavlik, Collabora

Overview

Due to the nature of the OpenXR interface, there is very little error information available to the developer and application. By using the `XR_EXT_debug_utils` extension, developers **can** obtain more information. When combined with validation layers, even more detailed feedback on the application's use of OpenXR will be provided.

This extension provides the following capabilities:

- The ability to create a debug messenger which will pass along debug messages to an application supplied callback.
- The ability to identify specific OpenXR handles using a name to improve tracking.

12.24.1. Object Debug Annotation

It can be useful for an application to provide its own content relative to a specific OpenXR handle.

Object Naming

xrSetDebugUtilsObjectNameEXT allows application developers to associate user-defined information with OpenXR handles.

This is useful when paired with the callback that you register when creating an XrDebugUtilsMessengerEXT object. When properly used, debug messages will contain not only the corresponding object handle, but the associated object name as well.

An application can change the name associated with an object simply by calling xrSetDebugUtilsObjectNameEXT again with a new string. If the objectName member of the XrDebugUtilsObjectNameInfoEXT structure is an empty string, then any previously set name is removed.

12.24.2. Debug Messengers

OpenXR allows an application to register arbitrary number of callbacks with all the OpenXR components wishing to report debug information. Some callbacks **can** log the information to a file, others **can** cause a debug break point or any other behavior defined by the application. A primary producer of callback messages are the validation layers. If the extension is enabled, an application **can** register callbacks even when no validation layers are enabled. The OpenXR loader, other layers, and runtimes **may** also produce callback messages.

The debug messenger will provide detailed feedback on the application's use of OpenXR when events of interest occur. When an event of interest does occur, the debug messenger will submit a debug message to the debug callback that was provided during its creation. Additionally, the debug messenger is responsible with filtering out debug messages that the callback isn't interested in and will only provide desired debug messages.

12.24.3. Debug Message Categorization

Messages that are triggered by the debug messenger are categorized by their message type and severity. Additionally, each message has a string value identifying its `messageId`. These 3 bits of information can be used to filter out messages so you only receive reports on the messages you desire. In fact, during debug messenger creation, the severity and type flag values are provided to indicate what messages should be allowed to trigger the user's callback.

Message Type

The message type indicates the general category the message falls under. Currently we have the following message types:

Table 4. XR_EXT_debug_utils Message Type Flag Descriptions

Enum	**Description**
`XR_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT`	Specifies a general purpose event type. This is typically a non-validation, non-performance event.
`XR_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT`	Specifies an event caused during a validation against the OpenXR specification that **may** indicate invalid OpenXR usage.
`XR_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT`	Specifies a potentially non-optimal use of OpenXR.

Enum	Description
XR_DEBUG_UTILS_MESSAGE_TYPE_CONFORMANCE_BIT_EXT	Specifies a non-conformant OpenXR result. This is typically caused by a layer or runtime returning non-conformant data.

A message may correspond to more than one type. For example, if a validation warning also could impact performance, then the message might be identified with both the XR_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT and XR_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT flag bits.

Message Severity

The severity of a message is a flag that indicates how important the message is using standard logging naming. The severity flag bit values are shown in the following table.

Table 5. XR_EXT_debug_utils Message Severity Flag Descriptions

Enum	Description
XR_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT	Specifies the most verbose output indicating all diagnostic messages from the OpenXR loader, layers, and drivers should be captured.
XR_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT	Specifies an informational message such as resource details that might be handy when debugging an application.
XR_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT	Specifies use of OpenXR that could be an application bug. Such cases **may** not be immediately harmful, such as providing too many swapchain images. Other cases **may** point to behavior that is almost certainly bad when unintended, such as using a swapchain image whose memory has not been filled. In general, if you see a warning but you know that the behavior is intended/desired, then simply ignore the warning.
XR_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT	Specifies an error that **may** cause undefined behavior, including an application crash.

> *Note*
>
> The values of XrDebugUtilsMessageSeverityFlagBitsEXT are sorted based on severity. The higher the flag value, the more severe the message. This allows for simple boolean operation comparisons when looking at XrDebugUtilsMessageSeverityFlagBitsEXT values.

Message IDs

The XrDebugUtilsMessengerCallbackDataEXT structure contains a `messageId` that may be a string identifying the message ID for the triggering debug message. This may be blank, or it may simply contain the name of an OpenXR component (like "OpenXR Loader"). However, when certain API layers or runtimes are used, especially the OpenXR core_validation API layer, then this value is intended to uniquely identify the message generated. If a certain warning/error message constantly fires, a user can simply look at the unique ID in their callback handler and manually filter it out.

For validation layers, this `messageId` value actually can be used to find the section of the OpenXR specification that the layer believes to have been violated. See the core_validation API Layer documentation for more information on how this can be done.

12.24.4. Session Labels

All OpenXR work is performed inside of an XrSession. There are times that it helps to label areas in your OpenXR session to allow easier debugging. This can be especially true if your application creates more than one session. There are two kinds of labels provided in this extension:

- Region labels
- Individual labels

To begin identifying a region using a debug label inside a session, you may use the xrSessionBeginDebugUtilsLabelRegionEXT function. Calls to xrSessionBeginDebugUtilsLabelRegionEXT may be nested allowing you to identify smaller and smaller labeled regions within your code. Using this, you can build a "call-stack" of sorts with labels since any logging callback will contain the list of all active session label regions.

To end the last session label region that was begun, you **must** call xrSessionEndDebugUtilsLabelRegionEXT. Each xrSessionBeginDebugUtilsLabelRegionEXT **must** have a matching xrSessionEndDebugUtilsLabelRegionEXT. All of a session's label region's **must** be closed by the xrDestroySession function is called for the given XrSession.

An individual debug label may be inserted at any time using xrSessionInsertDebugUtilsLabelEXT. The xrSessionInsertDebugUtilsLabelEXT is used to indicate a particular location within the execution of the application's session functions. The next call to xrSessionInsertDebugUtilsLabelEXT, xrSessionBeginDebugUtilsLabelRegionEXT, or xrSessionEndDebugUtilsLabelRegionEXT overrides this value.

New Object Types

```
XR_DEFINE_HANDLE(XrDebugUtilsMessengerEXT)
```

XrDebugUtilsMessengerEXT represents a callback function and associated filters registered with the runtime.

New Flag Types

```
typedef XrFlags64 XrDebugUtilsMessageSeverityFlagsEXT;
```

```
// Flag bits for XrDebugUtilsMessageSeverityFlagsEXT
static const XrDebugUtilsMessageSeverityFlagsEXT
XR_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT = 0x00000001;
static const XrDebugUtilsMessageSeverityFlagsEXT
XR_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT = 0x00000010;
static const XrDebugUtilsMessageSeverityFlagsEXT
XR_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT = 0x00000100;
static const XrDebugUtilsMessageSeverityFlagsEXT
XR_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT = 0x00001000;
```

```
typedef XrFlags64 XrDebugUtilsMessageTypeFlagsEXT;
```

```
// Flag bits for XrDebugUtilsMessageTypeFlagsEXT
static const XrDebugUtilsMessageTypeFlagsEXT XR_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT
= 0x00000001;
static const XrDebugUtilsMessageTypeFlagsEXT
XR_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT = 0x00000002;
static const XrDebugUtilsMessageTypeFlagsEXT
XR_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT = 0x00000004;
static const XrDebugUtilsMessageTypeFlagsEXT
XR_DEBUG_UTILS_MESSAGE_TYPE_CONFORMANCE_BIT_EXT = 0x00000008;
```

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT`
- `XR_TYPE_DEBUG_UTILS_MESSENGER_CALLBACK_DATA_EXT`

* XR_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT
* XR_TYPE_DEBUG_UTILS_LABEL_EXT

New Enums

New Structures

```
typedef struct XrDebugUtilsObjectNameInfoEXT {
    XrStructureType    type;
    const void*        next;
    XrObjectType       objectType;
    uint64_t           objectHandle;
    const char*        objectName;
} XrDebugUtilsObjectNameInfoEXT;
```

Member Descriptions

* `type` is the XrStructureType of this structure.
* `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
* `objectType` is an XrObjectType specifying the type of the object to be named.
* `objectHandle` is the object to be named.
* `objectName` is a `NULL` terminated UTF-8 string specifying the name to apply to objectHandle.

Valid Usage

* If `objectType` is `XR_OBJECT_TYPE_UNKNOWN`, `objectHandle` **must** not be XR_NULL_HANDLE
* If `objectType` is not `XR_OBJECT_TYPE_UNKNOWN`, `objectHandle` **must** be XR_NULL_HANDLE or an OpenXR handle of the type associated with `objectType`

Valid Usage (Implicit)

- The XR_EXT_debug_utils extension **must** be enabled prior to using XrDebugUtilsObjectNameInfoEXT
- type **must** be XR_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- objectType **must** be a valid XrObjectType value
- If objectName is not NULL, objectName **must** be a null-terminated UTF-8 string

```
typedef struct XrDebugUtilsLabelEXT {
    XrStructureType    type;
    const void*        next;
    const char*        labelName;
} XrDebugUtilsLabelEXT;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- labelName is a NULL terminated UTF-8 string specifying the label name.

Valid Usage (Implicit)

- The XR_EXT_debug_utils extension **must** be enabled prior to using XrDebugUtilsLabelEXT
- type **must** be XR_TYPE_DEBUG_UTILS_LABEL_EXT
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- labelName **must** be a null-terminated UTF-8 string

```
typedef struct XrDebugUtilsMessengerCallbackDataEXT {
    XrStructureType                  type;
    const void*                      next;
    const char*                      messageId;
    const char*                      functionName;
    const char*                      message;
    uint32_t                         objectCount;
    XrDebugUtilsObjectNameInfoEXT*   objects;
    uint32_t                         sessionLabelCount;
    XrDebugUtilsLabelEXT*            sessionLabels;
} XrDebugUtilsMessengerCallbackDataEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `messageId` is a `NULL` terminated string that identifies the message in a unique way. If the callback is triggered by a validation layer, this string corresponds the Valid Usage ID (VUID) that can be used to jump to the appropriate location in the OpenXR specification. This value **may** be `NULL` if no unique message identifier is associated with the message.
- `functionName` is a `NULL` terminated string that identifies the OpenXR function that was executing at the time the message callback was triggered. This value **may** be `NULL` in cases where it is difficult to determine the originating OpenXR function.
- `message` is a `NULL` terminated string detailing the trigger conditions.
- `objectCount` is a count of items contained in the `objects` array. This may be `0`.
- `objects` is a pointer to an array of XrDebugUtilsObjectNameInfoEXT objects related to the detected issue. The array is roughly in order or importance, but the 0th element is always guaranteed to be the most important object for this message.
- `sessionLabelCount` is a count of items contained in the `sessionLabels` array. This may be `0`.
- `sessionLabels` is a pointer to an array of XrDebugUtilsLabelEXT objects related to the detected issue. The array is roughly in order or importance, but the 0th element is always guaranteed to be the most important object for this message.
- `sessionLabels` is NULL or a pointer to an array of XrDebugUtilsLabelEXT active in the current XrSession at the time the callback was triggered. Refer to Session Labels for more information.

Valid Usage (Implicit)

- The `XR_EXT_debug_utils` extension **must** be enabled prior to using XrDebugUtilsMessengerCallbackDataEXT
- `type` **must** be `XR_TYPE_DEBUG_UTILS_MESSENGER_CALLBACK_DATA_EXT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `messageId` **must** be a null-terminated UTF-8 string
- `functionName` **must** be a null-terminated UTF-8 string
- `message` **must** be a null-terminated UTF-8 string

An XrDebugUtilsMessengerCallbackDataEXT is a messenger object that handles passing along debug messages to a provided debug callback.

> *Note*
>
> This structure should only be considered valid during the lifetime of the triggered callback.

The labels listed inside `sessionLabels` are organized in time order, with the most recently generated label appearing first, and the oldest label appearing last.

```
typedef struct XrDebugUtilsMessengerCreateInfoEXT {
    XrStructureType                         type;
    const void*                             next;
    XrDebugUtilsMessageSeverityFlagsEXT     messageSeverities;
    XrDebugUtilsMessageTypeFlagsEXT         messageTypes;
    PFN_xrDebugUtilsMessengerCallbackEXT    userCallback;
    void*                                   userData;
} XrDebugUtilsMessengerCreateInfoEXT;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- messageSeverities is a bitmask of XrDebugUtilsMessageSeverityFlagBitsEXT specifying which severity of event(s) that will cause this callback to be called.
- messageTypes is a combination of XrDebugUtilsMessageTypeFlagBitsEXT specifying which type of event(s) will cause this callback to be called.
- userCallback is the application defined callback function to call.
- userData is arbitrary user data to be passed to the callback.

Valid Usage

- userCallback **must** be a valid PFN_xrDebugUtilsMessengerCallbackEXT

Valid Usage (Implicit)

- The XR_EXT_debug_utils extension **must** be enabled prior to using XrDebugUtilsMessengerCreateInfoEXT
- type **must** be XR_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- messageSeverities **must** be a valid combination of XrDebugUtilsMessageSeverityFlagBitsEXT values
- messageSeverities **must** not be 0
- messageTypes **must** be a valid combination of XrDebugUtilsMessageTypeFlagBitsEXT values
- messageTypes **must** not be 0
- userCallback **must** be a valid PFN_xrDebugUtilsMessengerCallbackEXT value

For each XrDebugUtilsMessengerEXT that is created the XrDebugUtilsMessengerCreateInfoEXT::messageSeverities and XrDebugUtilsMessengerCreateInfoEXT::messageTypes determine when that XrDebugUtilsMessengerCreateInfoEXT::userCallback is called. The process to determine if the user's userCallback is triggered when an event occurs is as follows:

- The runtime will perform a bitwise AND of the event's XrDebugUtilsMessageSeverityFlagBitsEXT with the XrDebugUtilsMessengerCreateInfoEXT::messageSeverities provided during creation of the XrDebugUtilsMessengerEXT object.

- If this results in 0, the message is skipped.
- The runtime will perform bitwise AND of the event's XrDebugUtilsMessageTypeFlagBitsEXT with the XrDebugUtilsMessengerCreateInfoEXT::messageTypes provided during the creation of the XrDebugUtilsMessengerEXT object.
- If this results in 0, the message is skipped.
- If the message of the current event is not skipped, the callback will be called with the message.

The callback will come directly from the component that detected the event, unless some other layer intercepts the calls for its own purposes (filter them in a different way, log to a system error log, etc.).

An application **can** receive multiple callbacks if multiple XrDebugUtilsMessengerEXT objects are created. A callback will always be executed in the same thread as the originating OpenXR call.

> *Note*
>
> A callback **can** be called from multiple threads simultaneously if the application is making OpenXR calls from multiple threads.

New Functions

```
XrResult xrSetDebugUtilsObjectNameEXT(
    XrInstance                                  instance,
    const XrDebugUtilsObjectNameInfoEXT*        nameInfo);
```

Parameter Descriptions

- instance is the XrInstance that the object was created under.
- nameInfo is a pointer to an instance of the XrDebugUtilsObjectNameInfoEXT structure specifying the parameters of the name to set on the object.

Valid Usage

- In the structure pointed to by nameInfo, XrDebugUtilsObjectNameInfoEXT::objectType **must** not be XR_OBJECT_TYPE_UNKNOWN
- In the structure pointed to by nameInfo, XrDebugUtilsObjectNameInfoEXT::objectHandle **must** not be XR_NULL_HANDLE

Valid Usage (Implicit)

- The `XR_EXT_debug_utils` extension **must** be enabled prior to calling xrSetDebugUtilsObjectNameEXT
- `instance` **must** be a valid XrInstance handle
- `nameInfo` **must** be a pointer to a valid XrDebugUtilsObjectNameInfoEXT structure

Thread Safety

- Access to the `objectHandle` member of the `nameInfo` parameter **must** be externally synchronized

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_OUT_OF_MEMORY`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_FUNCTION_UNSUPPORTED`

Applications **may** change the name associated with an object simply by calling xrSetDebugUtilsObjectNameEXT again with a new string. If XrDebugUtilsObjectNameInfoEXT ::`objectName` is an empty string, then any previously set name is removed.

```
XrResult xrCreateDebugUtilsMessengerEXT(
    XrInstance                                  instance,
    const XrDebugUtilsMessengerCreateInfoEXT*   createInfo,
    XrDebugUtilsMessengerEXT*                   messenger);
```

Parameter Descriptions

- `instance` is the instance the messenger will be used with.
- `createInfo` points to an XrDebugUtilsMessengerCreateInfoEXT structure, which contains the callback pointer as well as defines the conditions under which this messenger will trigger the callback.
- `messenger` is a pointer to which the created XrDebugUtilsMessengerEXT object is returned.

Valid Usage (Implicit)

- The `XR_EXT_debug_utils` extension **must** be enabled prior to calling xrCreateDebugUtilsMessengerEXT
- `instance` **must** be a valid XrInstance handle
- `createInfo` **must** be a pointer to a valid XrDebugUtilsMessengerCreateInfoEXT structure
- `messenger` **must** be a pointer to an XrDebugUtilsMessengerEXT handle

Thread Safety

- Access to `instance`, and any child handles, **must** be externally synchronized

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_OUT_OF_MEMORY`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_LIMIT_REACHED`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_FUNCTION_UNSUPPORTED`

The application **must** ensure that xrCreateDebugUtilsMessengerEXT is not executed in parallel with any OpenXR function that is also called with `instance` or child of `instance`.

When an event of interest occurs a debug messenger calls its `createInfo`->`userCallback` with a debug message from the producer of the event. Additionally, the debug messenger **must** filter out any debug messages that the application's callback is not interested in based on XrDebugUtilsMessengerCreateInfoEXT flags, as described below.

```
XrResult xrDestroyDebugUtilsMessengerEXT(
    XrDebugUtilsMessengerEXT                    messenger);
```

Parameter Descriptions

- `messenger` the XrDebugUtilsMessengerEXT object to destroy. `messenger` is an externally synchronized object and **must** not be used on more than one thread at a time. This means that xrDestroyDebugUtilsMessengerEXT **must** not be called when a callback is active.

Valid Usage (Implicit)

- The `XR_EXT_debug_utils` extension **must** be enabled prior to calling xrDestroyDebugUtilsMessengerEXT
- `messenger` **must** be a valid XrDebugUtilsMessengerEXT handle

Thread Safety

- Access to `messenger` **must** be externally synchronized
- Access to the XrInstance used to create `messenger`, and all of its child handles **must** be externally synchronized

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_FUNCTION_UNSUPPORTED`

The application **must** ensure that xrDestroyDebugUtilsMessengerEXT is not executed in parallel with any OpenXR function that is also called with the `instance` or child of `instance` that it was created with.

```
XrResult                                    xrSubmitDebugUtilsMessageEXT(
    XrInstance                                  instance,
    XrDebugUtilsMessageSeverityFlagsEXT         messageSeverity,
    XrDebugUtilsMessageTypeFlagsEXT             messageTypes,
    const XrDebugUtilsMessengerCallbackDataEXT* callbackData);
```

Parameter Descriptions

- `instance` is the debug stream's XrInstance.
- `messageSeverity` is a single bit value of XrDebugUtilsMessageSeverityFlagsEXT severity of this event/message.
- `messageTypes` is an XrDebugUtilsMessageTypeFlagsEXT bitmask of XrDebugUtilsMessageTypeFlagBitsEXT specifying which types of event to identify this message with.
- `callbackData` contains all the callback related data in the XrDebugUtilsMessengerCallbackDataEXT structure.

Valid Usage

- For each structure in `objects` found in `callbackData`, the value of XrDebugUtilsObjectNameInfoEXT::`objectType` **must** not be `XR_OBJECT_TYPE_UNKNOWN`

Valid Usage (Implicit)

- The `XR_EXT_debug_utils` extension **must** be enabled prior to calling xrSubmitDebugUtilsMessageEXT
- `instance` **must** be a valid XrInstance handle
- `messageSeverity` **must** be a valid combination of XrDebugUtilsMessageSeverityFlagBitsEXT values
- `messageSeverity` **must** not be `0`
- `messageTypes` **must** be a valid combination of XrDebugUtilsMessageTypeFlagBitsEXT values
- `messageTypes` **must** not be `0`
- `callbackData` **must** be a pointer to a valid XrDebugUtilsMessengerCallbackDataEXT structure

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_FUNCTION_UNSUPPORTED

The application **can** also produce a debug message, and submit it into the OpenXR messaging system.

The call will propagate through the layers and generate callback(s) as indicated by the message's flags. The parameters are passed on to the callback in addition to the userData value that was defined at the time the messenger was created.

```
XrResult xrSessionBeginDebugUtilsLabelRegionEXT(
    XrSession                                   session,
    const XrDebugUtilsLabelEXT*                 labelInfo);
```

Parameter Descriptions

- session is the XrSession that a label region should be associated with.
- labelInfo is the XrDebugUtilsLabelEXT containing the label information for the region that should be begun.

Valid Usage (Implicit)

- The XR_EXT_debug_utils extension **must** be enabled prior to calling xrSessionBeginDebugUtilsLabelRegionEXT
- session **must** be a valid XrSession handle
- labelInfo **must** be a pointer to a valid XrDebugUtilsLabelEXT structure

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_FUNCTION_UNSUPPORTED

The xrSessionBeginDebugUtilsLabelRegionEXT function begins a label region within session.

```
XrResult xrSessionEndDebugUtilsLabelRegionEXT(
    XrSession                                   session);
```

Parameter Descriptions

- session is the XrSession that a label region should be associated with.

Valid Usage

- xrSessionEndDebugUtilsLabelRegionEXT **must** be called only after a matching xrSessionBeginDebugUtilsLabelRegionEXT.

Valid Usage (Implicit)

- The XR_EXT_debug_utils extension **must** be enabled prior to calling xrSessionEndDebugUtilsLabelRegionEXT
- session **must** be a valid XrSession handle

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_FUNCTION_UNSUPPORTED

This function ends the last label region begun with the xrSessionBeginDebugUtilsLabelRegionEXT function within the same session.

```
XrResult xrSessionInsertDebugUtilsLabelEXT(
    XrSession                                   session,
    const XrDebugUtilsLabelEXT*                 labelInfo);
```

Parameter Descriptions

- session is the XrSession that a label region should be associated with.
- labelInfo is the XrDebugUtilsLabelEXT containing the label information for the region that should be begun.

Valid Usage (Implicit)

- The XR_EXT_debug_utils extension **must** be enabled prior to calling xrSessionInsertDebugUtilsLabelEXT
- session **must** be a valid XrSession handle
- labelInfo **must** be a pointer to a valid XrDebugUtilsLabelEXT structure

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_FUNCTION_UNSUPPORTED

The xrSessionInsertDebugUtilsLabelEXT function inserts an individual label within session. The individual labels are useful for different reasons based on the type of debugging scenario. When used with something active like a profiler or debugger, it identifies a single point of time. When used with logging, the individual label identifies that a particular location has been passed at the point the log message is triggered. Because of this usage, individual labels only exist in a log until the next call to any of the label functions:

- xrSessionBeginDebugUtilsLabelRegionEXT
- xrSessionEndDebugUtilsLabelRegionEXT
- xrSessionInsertDebugUtilsLabelEXT

New Function Pointers

```
typedef XrBool32 (XRAPI_PTR *PFN_xrDebugUtilsMessengerCallbackEXT)(
            XrDebugUtilsMessageSeverityFlagsEXT              messageSeverity,
            XrDebugUtilsMessageTypeFlagsEXT                  messageTypes,
            const XrDebugUtilsMessengerCallbackDataEXT*      callbackData,
            void*                                            userData);
```

Parameter Descriptions

- `messageSeverity` indicates the single bit value of XrDebugUtilsMessageSeverityFlagsEXT that triggered this callback.
- `messageTypes` indicates the XrDebugUtilsMessageTypeFlagsEXT specifying which types of event triggered this callback.
- `callbackData` contains all the callback related data in the XrDebugUtilsMessengerCallbackDataEXT structure.
- `userData` is the user data provided when the XrDebugUtilsMessengerEXT was created.

The callback **must** not call xrDestroyDebugUtilsMessengerEXT.

The callback returns an `XrBool32` that indicates to the calling layer the application's desire to abort the call. A value of `XR_TRUE` indicates that the application wants to abort this call. If the application returns `XR_FALSE`, the function **must** not be aborted. Applications **should** always return `XR_FALSE` so that they see the same behavior with and without validation layers enabled.

If the application returns `XR_TRUE` from its callback and the OpenXR call being aborted returns an XrResult, the layer will return `XR_ERROR_VALIDATION_FAILURE`.

The object pointed to by `callbackData` (and any pointers in it recursively) **must** be valid during the lifetime of the triggered callback. It **may** become invalid afterwards.

Examples

Example 1

XR_EXT_debug_utils allows an application to register multiple callbacks with any OpenXR component wishing to report debug information. Some callbacks may log the information to a file, others may cause a debug break point or other application defined behavior. An application **can** register callbacks even when no validation layers are enabled, but they will only be called for loader and, if implemented, driver events.

To capture events that occur while creating or destroying an instance an application **can** link an XrDebugUtilsMessengerCreateInfoEXT structure to the next element of the XrInstanceCreateInfo structure given to xrCreateInstance. This callback is only valid for the duration of the xrCreateInstance and the xrDestroyInstance call. Use xrCreateDebugUtilsMessengerEXT to create persistent callback objects.

Example uses: Create three callback objects. One will log errors and warnings to the debug console using Windows `OutputDebugString`. The second will cause the debugger to break at that callback when an error happens and the third will log warnings to stdout.

```
extern XrInstance instance;  // previously initialized
```

```
// Must call extension functions through a function pointer:
PFN_xrCreateDebugUtilsMessengerEXT pfnCreateDebugUtilsMessengerEXT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrCreateDebugUtilsMessengerEXT",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                                 &pfnCreateDebugUtilsMessengerEXT)));

PFN_xrDestroyDebugUtilsMessengerEXT pfnDestroyDebugUtilsMessengerEXT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrDestroyDebugUtilsMessengerEXT",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                                 &pfnDestroyDebugUtilsMessengerEXT)));

XrDebugUtilsMessengerCreateInfoEXT callback1 = {
    XR_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT,   // type
    NULL,                                            // next
    XR_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT |  // messageSeverities
        XR_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT,
    XR_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |  // messageTypes
        XR_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT,
    myOutputDebugString,  // userCallback
    NULL                  // userData
};
XrDebugUtilsMessengerEXT messenger1 = XR_NULL_HANDLE;
CHK_XR(pfnCreateDebugUtilsMessengerEXT(instance, &callback1, &messenger1));

callback1.messageSeverities = XR_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
callback1.userCallback = myDebugBreak;
callback1.userData = NULL;
XrDebugUtilsMessengerEXT messenger2 = XR_NULL_HANDLE;
CHK_XR(pfnCreateDebugUtilsMessengerEXT(instance, &callback1, &messenger2));

XrDebugUtilsMessengerCreateInfoEXT callback3 = {
    XR_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT,    // type
    NULL,                                             // next
    XR_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT,  // messageSeverities
    XR_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |     // messageTypes
        XR_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT,
    myStdOutLogger,  // userCallback
    NULL             // userData
};
XrDebugUtilsMessengerEXT messenger3 = XR_NULL_HANDLE;
CHK_XR(pfnCreateDebugUtilsMessengerEXT(instance, &callback3, &messenger3));

// ...

// Remove callbacks when cleaning up
pfnDestroyDebugUtilsMessengerEXT(messenger1);
pfnDestroyDebugUtilsMessengerEXT(messenger2);
```

```
pfnDestroyDebugUtilsMessengerEXT(messenger3);
```

Example 2

Associate a name with an XrSpace, for easier debugging in external tools or with validation layers that can print a friendly name when referring to objects in error messages.

```
extern XrInstance instance;  // previously initialized
extern XrSpace space;        // previously initialized

// Must call extension functions through a function pointer:
PFN_xrSetDebugUtilsObjectNameEXT pfnSetDebugUtilsObjectNameEXT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrSetDebugUtilsObjectNameEXT",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                                 &pfnSetDebugUtilsObjectNameEXT)));

// Set a name on the space
const XrDebugUtilsObjectNameInfoEXT spaceNameInfo = {
    XR_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT,  // type
    NULL,                                      // next
    XR_OBJECT_TYPE_SPACE,                      // objectType
    (uint64_t)space,                           // objectHandle
    "My Object-Specific Space",                // objectName
};

pfnSetDebugUtilsObjectNameEXT(instance, &spaceNameInfo);

// A subsequent error might print:
//   Space "My Object-Specific Space" (0xc0dec0dedeadbeef) is used
//   with an XrSession that is not it's parent.
```

Example 3

Labeling the workload with naming information so that any form of analysis can display a more usable visualization of where actions occur in the lifetime of a session.

```
extern XrInstance instance;  // previously initialized
extern XrSession session;    // previously initialized

// Must call extension functions through a function pointer:

PFN_xrSessionBeginDebugUtilsLabelRegionEXT pfnSessionBeginDebugUtilsLabelRegionEXT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrSessionBeginDebugUtilsLabelRegionEXT",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                                 &pfnSessionBeginDebugUtilsLabelRegionEXT)));
```

```
PFN_xrSessionEndDebugUtilsLabelRegionEXT pfnSessionEndDebugUtilsLabelRegionEXT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrSessionEndDebugUtilsLabelRegionEXT",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                                 &pfnSessionEndDebugUtilsLabelRegionEXT)));

PFN_xrSessionInsertDebugUtilsLabelEXT pfnSessionInsertDebugUtilsLabelEXT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrSessionInsertDebugUtilsLabelEXT",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                                 &pfnSessionInsertDebugUtilsLabelEXT)));

XrSessionBeginInfo session_begin_info = {
    XR_TYPE_SESSION_BEGIN_INFO,
    nullptr,
    XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO
};
xrBeginSession(session, &session_begin_info);

const XrDebugUtilsLabelEXT session_active_region_label = {
    XR_TYPE_DEBUG_UTILS_LABEL_EXT,  // type
    NULL,                           // next
    "Session active",               // labelName
};

// Start an annotated region of calls under the 'Session Active' name
pfnSessionBeginDebugUtilsLabelRegionEXT(session, &session_active_region_label);

// Brackets added for clarity
{
    XrDebugUtilsLabelEXT individual_label = {
        XR_TYPE_DEBUG_UTILS_LABEL_EXT,  // type
        NULL,                           // next
        "WaitFrame",                    // labelName
    };

    const char wait_frame_label[] = "WaitFrame";
    individual_label.labelName = wait_frame_label;
    pfnSessionInsertDebugUtilsLabelEXT(session, &individual_label);
    XrFrameWaitInfo wait_frame_info;  // initialization omitted for readability
    XrFrameState frame_state = {XR_TYPE_FRAME_STATE, nullptr};
    xrWaitFrame(session, &wait_frame_info, &frame_state);

    // Do stuff 1

    const XrDebugUtilsLabelEXT session_frame_region_label = {
        XR_TYPE_DEBUG_UTILS_LABEL_EXT,  // type
        NULL,                           // next
        "Session Frame 123",            // labelName
```

```
        };

        // Start an annotated region of calls under the 'Session Frame 123' name
        pfnSessionBeginDebugUtilsLabelRegionEXT(session, &session_frame_region_label);

        // Brackets added for clarity
        {

            const char begin_frame_label[] = "BeginFrame";
            individual_label.labelName = begin_frame_label;
            pfnSessionInsertDebugUtilsLabelEXT(session, &individual_label);

            XrFrameBeginInfo begin_frame_info;  // initialization omitted for readability
            xrBeginFrame(session, &begin_frame_info);

            // Do stuff 2

            const char end_frame_label[] = "EndFrame";
            individual_label.labelName = end_frame_label;
            pfnSessionInsertDebugUtilsLabelEXT(session, &individual_label);

            XrFrameEndInfo end_frame_info;  // initialization omitted for readability
            xrEndFrame(session, &end_frame_info);
        }

        // End the session/begun region started above
        // (in this case it's the "Session Frame 123" label)
        pfnSessionEndDebugUtilsLabelRegionEXT(session);
    }

    // End the session/begun region started above
    // (in this case it's the "Session Active" label)
    pfnSessionEndDebugUtilsLabelRegionEXT(session);

    xrEndSession(session);
```

In the above example, if an error occurred in the `// Do stuff 1` section, then your debug utils callback would contain the following data in its `sessionLabels` array:

- `[0]` = `individual_label` with `labelName` = "WaitFrame"
- `[1]` = `session_active_region_label` with `labelName` = "Session active"

However, if an error occurred in the `// Do stuff 2` section, then your debug utils callback would contain the following data in its `sessionLabels` array:

- `[0]` = `individual_label` with `labelName` = "BeginFrame"

* `[1]` = `session_frame_region_label` with `labelName` = "Session Frame 123"
* `[2]` = `session_active_region_label` with `labelName` = "Session active"

You'll notice that "WaitFrame" is no longer available as soon as the next call to another function like xrSessionBeginDebugUtilsLabelRegionEXT.

Issues

None

Version History

* Revision 1, 2018-02-19 (Mark Young / Karl Schultz)
  * Initial draft, based on VK_EXT_debug_utils.
* Revision 2, 2018-11-16 (Mark Young)
  * Clean up some language based on changes going into the Vulkan VK_EXT_debug_utils extension by Peter Kraus (aka @krOoze).
  * Added session labels
* Revision 3, 2019-07-19 (Ryan Pavlik)
  * Update examples.
  * Improve formatting.
* Revision 4, 2021-04-04 (Ryan Pavlik)
  * Fix missing error code.
  * Improve formatting.

12.25. XR_EXT_eye_gaze_interaction

Name String

`XR_EXT_eye_gaze_interaction`

Extension Type

Instance extension

Registered Extension Number

31

Revision

1

Extension and Version Dependencies

* Requires OpenXR 1.0

Last Modified Date

2020-02-20

IP Status

No known IP claims.

Contributors

Denny Rönngren, Tobii
Yin Li, Microsoft
Alex Turner, Microsoft
Paul Pedriana, Oculus
Rémi Arnaud, Varjo
Blake Taylor, Magic Leap
Lachlan Ford, Microsoft
Cass Everitt, Oculus

Overview

This extension provides a `XrPath` for getting eye gaze input from an eye tracker to enable eye gaze interactions.

The intended use for this extension is to provide:

- system properties to inform if eye gaze interaction is supported by the current device.
- a `XrPath` for real time eye tracking that exposes an accurate and precise eye gaze pose to be used to enable eye gaze interactions.
- a structure XrEyeGazeSampleTimeEXT that allows for an application to retrieve more information regarding the eye tracking samples.

With these building blocks, an application can discover if the XR runtime has access to an eye tracker, bind the eye gaze pose to the action system, determine if the eye tracker is actively tracking the users eye gaze, and use the eye gaze pose as an input signal to build eye gaze interactions.

12.25.1. Eye tracker

An eye tracker is a sensory device that tracks eyes and accurately maps what the user is looking at. The main purpose of this extension is to provide accurate and precise eye gaze for the application.

Eye tracking data can be sensitive personal information and is closely linked to personal privacy and integrity. It is strongly recommended that applications that store or transfer eye tracking data always ask the user for active and specific acceptance to do so.

If a runtime supports a permission system to control application access to the eye tracker, then the runtime **must** set the `isActive` field to `XR_FALSE` on the supplied XrActionStatePose structure, and **must** clear `XR_SPACE_LOCATION_POSITION_TRACKED_BIT`, `XR_SPACE_LOCATION_POSITION_VALID_BIT`,

XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT and XR_SPACE_LOCATION_ORIENTATION_VALID_BIT when locating using the tracked space until the application has been allowed access to the eye tracker. When the application access has been allowed, the runtime **may** set isActive on the supplied XrActionStatePose structure to XR_TRUE and **may** set XR_SPACE_LOCATION_POSITION_TRACKED_BIT, XR_SPACE_LOCATION_POSITION_VALID_BIT XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT and XR_SPACE_LOCATION_ORIENTATION_VALID_BIT when locating using the tracked space.

12.25.2. Device enumeration

When the eye gaze input extension is enabled an application **may** pass in a XrSystemEyeGazeInteractionPropertiesEXT structure in next chain structure when calling xrGetSystemProperties to acquire information about the connected eye tracker.

The runtime **must** populate the XrSystemEyeGazeInteractionPropertiesEXT structure with the relevant information to the XrSystemProperties returned by the xrGetSystemProperties call.

```
typedef struct XrSystemEyeGazeInteractionPropertiesEXT {
    XrStructureType    type;
    void*              next;
    XrBool32           supportsEyeGazeInteraction;
} XrSystemEyeGazeInteractionPropertiesEXT;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- supportsEyeGazeInteraction the runtime **must** set this value to XR_TRUE when eye gaze sufficient for use cases such as aiming or targeting is supported by the current device, otherwise the runtime **must** set this to XR_FALSE.

Valid Usage (Implicit)

- The XR_EXT_eye_gaze_interaction extension **must** be enabled prior to using XrSystemEyeGazeInteractionPropertiesEXT
- type **must** be XR_TYPE_SYSTEM_EYE_GAZE_INTERACTION_PROPERTIES_EXT
- next **must** be NULL or a valid pointer to the next structure in a structure chain

12.25.3. Eye gaze input

This extension exposes a new interaction profile path */interaction_profiles/ext/eye_gaze_interaction* that is valid for the user path

- */user/eyes_ext*

for supported input source

- *…/input/gaze_ext/pose*

The eye gaze pose is natively oriented with +Y up, +X to the right, and -Z forward and not gravity-aligned, similar to the XR_REFERENCE_SPACE_TYPE_VIEW. The eye gaze pose may originate from a point positioned between the user's eyes. At any point of time both the position and direction of the eye pose is tracked or untracked. This means that the runtime **must** set both XR_SPACE_LOCATION_POSITION_TRACKED_BIT and XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT or clear both XR_SPACE_LOCATION_POSITION_TRACKED_BIT and XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT.

One particularity for eye trackers compared to most other spatial input is that the runtime may not have the capability to predict or interpolate eye gaze poses. Runtimes that cannot predict or interpolate eye gaze poses **must** clamp the gaze pose requested in the xrLocateSpace call to the value nearest to time requested in the call. To allow for an application to reason about high accuracy eye tracking, the application **can** chain in an XrEyeGazeSampleTimeEXT to the next pointer of the XrSpaceLocation structure passed into the xrLocateSpace call. The runtime **must** set time in the XrEyeGazeSampleTimeEXT structure to the clamped, predicted or interpolated time. The application **should** inspect the time field to understand when in time the pose is expressed. The time field **may** be in the future if a runtime can predict gaze poses. If an XrEyeGazeSampleTimeEXT structure is passed into the xrLocateSpace call, and while neither space or baseSpace are bound to an action with toplevel path */user/eyes_ext/* the runtime **must** return XR_ERROR_VALIDATION_FAILURE.

When the runtime provides a nominal eye gaze pose, the XR_SPACE_LOCATION_POSITION_TRACKED_BIT **must** be set if the eye otherwise has a fully-tracked pose relative to the other space. A runtime **can** provide a sub-nominal eye-gaze pose but **must** then clear the XR_SPACE_LOCATION_POSITION_TRACKED_BIT. An application can expect that a nominal eye gaze pose can be used for use cases such as aiming or targeting, while a sub-nominal eye gaze pose has degraded performance and should not be relied on for all input scenarios. Applications should be very careful when using sub-nominal eye gaze pose, since the behaviour can vary considerably for different users and manufacturers, and some manufacturers **may** not provide sub-nominal eye gaze pose at all.

With current technology, some eye trackers **may** need to undergo an explicit calibration routine to provide a nominal accurate and precise eye gaze pose. If the eye tracker is in an uncalibrated state when the first call to xrSyncActions is made with an eye gaze action enabled, then the runtime **should** request eye tracker calibration from the user if it has not yet been requested.

```
typedef struct XrEyeGazeSampleTimeEXT {
    XrStructureType    type;
    void*              next;
    XrTime             time;
} XrEyeGazeSampleTimeEXT;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- time is when in time the eye gaze pose is expressed.

Valid Usage (Implicit)

- The XR_EXT_eye_gaze_interaction extension **must** be enabled prior to using XrEyeGazeSampleTimeEXT
- type **must** be XR_TYPE_EYE_GAZE_SAMPLE_TIME_EXT
- next **must** be NULL or a valid pointer to the next structure in a structure chain

12.25.4. Sample code

The following example code shows how to bind the eye pose to the action system.

```
extern XrInstance instance;
extern XrSession session;
extern XrPosef pose_identity;

// Create action set
XrActionSetCreateInfo actionSetInfo{XR_TYPE_ACTION_SET_CREATE_INFO};
strcpy(actionSetInfo.actionSetName, "gameplay");
strcpy(actionSetInfo.localizedActionSetName, "Gameplay");
actionSetInfo.priority = 0;
XrActionSet gameplayActionSet;
CHK_XR(xrCreateActionSet(instance, &actionSetInfo, &gameplayActionSet));

// Create user intent action
XrActionCreateInfo actionInfo{XR_TYPE_ACTION_CREATE_INFO};
strcpy(actionInfo.actionName, "user_intent");
actionInfo.actionType = XR_ACTION_TYPE_POSE_INPUT;
```

```
strcpy(actionInfo.localizedActionName, "User Intent");
XrAction userIntentAction;
CHK_XR(xrCreateAction(gameplayActionSet, &actionInfo, &userIntentAction));

// Create suggested bindings
XrPath eyeGazeInteractionProfilePath;
CHK_XR(xrStringToPath(instance, "/interaction_profiles/ext/eye_gaze_interaction",
&eyeGazeInteractionProfilePath));

XrPath gazePosePath;
CHK_XR(xrStringToPath(instance, "/user/eyes_ext/input/gaze_ext/pose", &gazePosePath));

XrActionSuggestedBinding bindings;
bindings.action = userIntentAction;
bindings.binding = gazePosePath;

XrInteractionProfileSuggestedBinding
suggestedBindings{XR_TYPE_INTERACTION_PROFILE_SUGGESTED_BINDING};
suggestedBindings.interactionProfile = eyeGazeInteractionProfilePath;
suggestedBindings.suggestedBindings = &bindings;
suggestedBindings.countSuggestedBindings = 1;
CHK_XR(xrSuggestInteractionProfileBindings(instance, &suggestedBindings));

XrSessionActionSetsAttachInfo attachInfo{XR_TYPE_SESSION_ACTION_SETS_ATTACH_INFO};
attachInfo.countActionSets = 1;
attachInfo.actionSets = &gameplayActionSet;
CHK_XR(xrAttachSessionActionSets(session, &attachInfo));

XrActionSpaceCreateInfo createActionSpaceInfo{XR_TYPE_ACTION_SPACE_CREATE_INFO};
createActionSpaceInfo.action = userIntentAction;
createActionSpaceInfo.poseInActionSpace = pose_identity;
XrSpace gazeActionSpace;
CHK_XR(xrCreateActionSpace(session, &createActionSpaceInfo, &gazeActionSpace));

XrReferenceSpaceCreateInfo createReferenceSpaceInfo{XR_TYPE_REFERENCE_SPACE_CREATE_INFO};
createReferenceSpaceInfo.referenceSpaceType = XR_REFERENCE_SPACE_TYPE_LOCAL;
createReferenceSpaceInfo.poseInReferenceSpace = pose_identity;
XrSpace localReferenceSpace;
CHK_XR(xrCreateReferenceSpace(session, &createReferenceSpaceInfo, &localReferenceSpace));

while(true)
{
  XrActiveActionSet activeActionSet{gameplayActionSet, XR_NULL_PATH};
  XrTime time;

  XrActionsSyncInfo syncInfo{XR_TYPE_ACTIONS_SYNC_INFO};
  syncInfo.countActiveActionSets = 1;
  syncInfo.activeActionSets = &activeActionSet;
```

```
  CHK_XR(xrSyncActions(session, &syncInfo));

  XrActionStatePose actionStatePose{XR_TYPE_ACTION_STATE_POSE};
  XrActionStateGetInfo getActionStateInfo{XR_TYPE_ACTION_STATE_GET_INFO};
  getActionStateInfo.action = userIntentAction;
  CHK_XR(xrGetActionStatePose(session, &getActionStateInfo, &actionStatePose));

  if(actionStatePose.isActive){
    XrEyeGazeSampleTimeEXT eyeGazeSampleTime{XR_TYPE_EYE_GAZE_SAMPLE_TIME_EXT};
    XrSpaceLocation gazeLocation{XR_TYPE_SPACE_LOCATION, &eyeGazeSampleTime};
    CHK_XR(xrLocateSpace(gazeActionSpace, localReferenceSpace, time, &gazeLocation));

    // Do things
  }
}
```

Version History

- Revision 1, 2020-02-20 (Denny Rönngren)
  - Initial version

12.26. XR_EXT_hand_joints_motion_range

Name String

XR_EXT_hand_joints_motion_range

Extension Type

Instance extension

Registered Extension Number

81

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0
- Requires XR_EXT_hand_tracking

Last Modified Date

2021-04-15

IP Status

No known IP claims.

Contributors

Joe van den Heuvel, Valve
Rune Berg, Valve
Joe Ludwig, Valve

Overview

This extension augments the `XR_EXT_hand_tracking` extension to enable applications to request that the XrHandJointLocationsEXT returned by xrLocateHandJointsEXT should return hand joint locations conforming to a range of motion specified by the application.

The application **must** enable the `XR_EXT_hand_tracking` extension in order to use this extension.

New Object Types

New Flag Types

New Enum Constants

New Enums

The XrHandJointsMotionRangeEXT describes the hand joints' range of motion returned by xrLocateHandJointsEXT.

Runtimes **must** support both `XR_HAND_JOINTS_MOTION_RANGE_CONFORMING_TO_CONTROLLER_EXT` and `XR_HAND_JOINTS_MOTION_RANGE_UNOBSTRUCTED_EXT` for each controller interaction profile that supports hand joint data.

```
typedef enum XrHandJointsMotionRangeEXT {
    XR_HAND_JOINTS_MOTION_RANGE_UNOBSTRUCTED_EXT = 1,
    XR_HAND_JOINTS_MOTION_RANGE_CONFORMING_TO_CONTROLLER_EXT = 2,
    XR_HAND_JOINTS_MOTION_RANGE_MAX_ENUM_EXT = 0x7FFFFFFF
} XrHandJointsMotionRangeEXT;
```

Enumerant Descriptions

- `XR_HAND_JOINTS_MOTION_RANGE_UNOBSTRUCTED_EXT` This option refers to the range of motion of a human hand, without any obstructions. Input systems that obstruct the movement of the user's hand (eg: a held controller preventing the user from making a fist) or have only limited ability to track finger positions **must** use the information available to them to emulate an unobstructed range of motion.
- `XR_HAND_JOINTS_MOTION_RANGE_CONFORMING_TO_CONTROLLER_EXT` This option refers to the range of motion of the hand joints taking into account any physical limits imposed by the controller itself. This will tend to be the most accurate pose compared to the user's actual hand pose, but might not allow a closed fist for example.
  - If the current interaction profile represents a controller, or other device that obstructs the hand, the implementation **must** return joint locations conforming to the shape of that device. If the current interaction profile is being emulated by a different physical controller, the implementation **may** return joint locations conforming to the shape of either the current interaction profile or the actual physical controller.
  - If the current interaction profile does not represent a controller, the implementation **must** return joint locations based on the unobstructed joint locations.

New Structures

The XrHandJointsMotionRangeInfoEXT is a structure that an application **can** chain in XrHandJointsLocateInfoEXT to request the joint motion range specified by the `handJointsMotionRange` field.

Runtimes **must** return the appropriate joint locations depending on the `handJointsMotionRange` field and the currently active interaction profile.

```
typedef struct XrHandJointsMotionRangeInfoEXT {
    XrStructureType               type;
    const void*                   next;
    XrHandJointsMotionRangeEXT    handJointsMotionRange;
} XrHandJointsMotionRangeInfoEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `handJointsMotionRange` is an XrHandJointsMotionRangeEXT that defines the hand joint range of motion the application wants.

Valid Usage (Implicit)

- The `XR_EXT_hand_joints_motion_range` extension **must** be enabled prior to using XrHandJointsMotionRangeInfoEXT
- `type` **must** be `XR_TYPE_HAND_JOINTS_MOTION_RANGE_INFO_EXT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `handJointsMotionRange` **must** be a valid XrHandJointsMotionRangeEXT value

New Functions

Issues

Version History

- Revision 1, 2021-04-15 (Rune Berg)
  - Initial extension description

12.27. XR_EXT_hand_tracking

Name String

`XR_EXT_hand_tracking`

Extension Type

Instance extension

Registered Extension Number

52

Revision

4

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2021-04-15

IP Status

No known IP claims.

Contributors

Yin Li, Microsoft
Lachlan Ford, Microsoft
Alex Turner, Microsoft
Bryce Hutchings, Microsoft
Cass Everitt, Oculus
Blake Taylor, Magic Leap
Joe van den Heuvel, Valve
Rune Berg, Valve
Nicholas Benson, Ultraleap
Ryan Pavlik, Collabora

Overview

This extension enables applications to locate the individual joints of hand tracking inputs. It enables applications to render hands in XR experiences and interact with virtual objects using hand joints.

Inspect system capability

An application **can** inspect whether the system is capable of hand tracking input by extending the XrSystemProperties with XrSystemHandTrackingPropertiesEXT structure when calling xrGetSystemProperties.

```
typedef struct XrSystemHandTrackingPropertiesEXT {
    XrStructureType    type;
    void*              next;
    XrBool32           supportsHandTracking;
} XrSystemHandTrackingPropertiesEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `supportsHandTracking` is an `XrBool32`, indicating if current system is capable of hand tracking input.

Valid Usage (Implicit)

- The `XR_EXT_hand_tracking` extension **must** be enabled prior to using XrSystemHandTrackingPropertiesEXT
- `type` **must** be `XR_TYPE_SYSTEM_HAND_TRACKING_PROPERTIES_EXT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

If a runtime returns `XR_FALSE` for `supportsHandTracking`, the runtime **must** return `XR_ERROR_FEATURE_UNSUPPORTED` from xrCreateHandTrackerEXT.

Create a hand tracker handle

The XrHandTrackerEXT handle represents the resources for hand tracking of the specific hand.

```
XR_DEFINE_HANDLE(XrHandTrackerEXT)
```

An application creates separate XrHandTrackerEXT handles for left and right hands. This handle can be used to locate hand joints using xrLocateHandJointsEXT function.

A hand tracker provides joint locations with an unobstructed range of motion of an empty human hand.

> *Note*
>
> This behavior can be modified by the `XR_EXT_hand_joints_motion_range` extension

An application can create an XrHandTrackerEXT handle using xrCreateHandTrackerEXT function.

```
XrResult xrCreateHandTrackerEXT(
    XrSession                                   session,
    const XrHandTrackerCreateInfoEXT*           createInfo,
    XrHandTrackerEXT*                           handTracker);
```

Parameter Descriptions

- `session` is an XrSession in which the hand tracker will be active.
- `createInfo` is the XrHandTrackerCreateInfoEXT used to specify the hand tracker.
- `handTracker` is the returned XrHandTrackerEXT handle.

Valid Usage (Implicit)

- The `XR_EXT_hand_tracking` extension **must** be enabled prior to calling xrCreateHandTrackerEXT
- `session` **must** be a valid XrSession handle
- `createInfo` **must** be a pointer to a valid XrHandTrackerCreateInfoEXT structure
- `handTracker` **must** be a pointer to an XrHandTrackerEXT handle

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_FEATURE_UNSUPPORTED`

If the system does not support hand tracking, runtime **must** return `XR_ERROR_FEATURE_UNSUPPORTED` from xrCreateHandTrackerEXT. In this case, the runtime **must** return `XR_FALSE` for `supportsHandTracking` in

XrSystemHandTrackingPropertiesEXT when the function xrGetSystemProperties is called, so that the application **can** avoid creating a hand tracker.

The XrHandTrackerCreateInfoEXT structure describes the information to create a XrHandTrackerEXT handle.

```
typedef struct XrHandTrackerCreateInfoEXT {
    XrStructureType      type;
    const void*          next;
    XrHandEXT            hand;
    XrHandJointSetEXT    handJointSet;
} XrHandTrackerCreateInfoEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `hand` is an XrHandEXT which describes which hand the tracker is tracking.
- `handJointSet` is an XrHandJointSetEXT describe the set of hand joints to retrieve.

Valid Usage (Implicit)

- The `XR_EXT_hand_tracking` extension **must** be enabled prior to using XrHandTrackerCreateInfoEXT
- `type` **must** be `XR_TYPE_HAND_TRACKER_CREATE_INFO_EXT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrHandPoseTypeInfoMSFT
- `hand` **must** be a valid XrHandEXT value
- `handJointSet` **must** be a valid XrHandJointSetEXT value

The XrHandEXT describes which hand the XrHandTrackerEXT is tracking.

```
typedef enum XrHandEXT {
    XR_HAND_LEFT_EXT = 1,
    XR_HAND_RIGHT_EXT = 2,
    XR_HAND_MAX_ENUM_EXT = 0x7FFFFFFF
} XrHandEXT;
```

Enumerant Descriptions

- `XR_HAND_LEFT_EXT` specifies the hand tracker will be tracking the user's left hand.
- `XR_HAND_RIGHT_EXT` specifies the hand tracker will be tracking the user's right hand.

The XrHandJointSetEXT enum describes the set of hand joints to track when creating an XrHandTrackerEXT.

```
typedef enum XrHandJointSetEXT {
    XR_HAND_JOINT_SET_DEFAULT_EXT = 0,
    XR_HAND_JOINT_SET_MAX_ENUM_EXT = 0x7FFFFFFF
} XrHandJointSetEXT;
```

Enumerant Descriptions

- `XR_HAND_JOINT_SET_DEFAULT_EXT` indicates that the created XrHandTrackerEXT tracks the set of hand joints described by XrHandJointEXT enum, i.e. the xrLocateHandJointsEXT function returns an array of joint locations with the count of XR_HAND_JOINT_COUNT_EXT and can be indexed using XrHandJointEXT.

xrDestroyHandTrackerEXT function releases the `handTracker` and the underlying resources when finished with hand tracking experiences.

```
XrResult xrDestroyHandTrackerEXT(
    XrHandTrackerEXT                            handTracker);
```

Parameter Descriptions

- `handTracker` is an XrHandTrackerEXT previously created by xrCreateHandTrackerEXT.

Valid Usage (Implicit)

- The XR_EXT_hand_tracking extension **must** be enabled prior to calling xrDestroyHandTrackerEXT
- handTracker **must** be a valid XrHandTrackerEXT handle

Thread Safety

- Access to handTracker, and any child handles, **must** be externally synchronized

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_HANDLE_INVALID
- XR_ERROR_FUNCTION_UNSUPPORTED

Locate hand joints

The xrLocateHandJointsEXT function locates an array of hand joints to a base space at given time.

```
XrResult xrLocateHandJointsEXT(
    XrHandTrackerEXT                            handTracker,
    const XrHandJointsLocateInfoEXT*            locateInfo,
    XrHandJointLocationsEXT*                    locations);
```

Parameter Descriptions

- handTracker is an XrHandTrackerEXT previously created by xrCreateHandTrackerEXT.
- locateInfo is a pointer to XrHandJointsLocateInfoEXT describing information to locate hand joints.
- locations is a pointer to XrHandJointLocationsEXT receiving the returned hand joint locations.

Valid Usage (Implicit)

- The XR_EXT_hand_tracking extension **must** be enabled prior to calling xrLocateHandJointsEXT
- handTracker **must** be a valid XrHandTrackerEXT handle
- locateInfo **must** be a pointer to a valid XrHandJointsLocateInfoEXT structure
- locations **must** be a pointer to an XrHandJointLocationsEXT structure

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_TIME_INVALID
- XR_ERROR_FUNCTION_UNSUPPORTED

The XrHandJointsLocateInfoEXT structure describes the information to locate hand joints.

```
typedef struct XrHandJointsLocateInfoEXT {
    XrStructureType    type;
    const void*        next;
    XrSpace            baseSpace;
    XrTime             time;
} XrHandJointsLocateInfoEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `baseSpace` is an XrSpace within which the returned hand joint locations will be represented.
- `time` is an `XrTime` at which to locate the hand joints.

Valid Usage (Implicit)

- The `XR_EXT_hand_tracking` extension **must** be enabled prior to using XrHandJointsLocateInfoEXT
- `type` **must** be `XR_TYPE_HAND_JOINTS_LOCATE_INFO_EXT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrHandJointsMotionRangeInfoEXT
- `baseSpace` **must** be a valid XrSpace handle

XrHandJointLocationsEXT structure returns the state of the hand joint locations.

```
typedef struct XrHandJointLocationsEXT {
    XrStructureType             type;
    void*                       next;
    XrBool32                    isActive;
    uint32_t                    jointCount;
    XrHandJointLocationEXT*     jointLocations;
} XrHandJointLocationsEXT;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain, such as XrHandJointVelocitiesEXT.
- isActive is a XrBool32 indicating if the hand tracker is actively tracking.
- jointCount is a uint32_t describing the count of elements in jointLocations array.
- jointLocations is an array of XrHandJointLocationEXT receiving the returned hand joint locations.

The application **must** allocate the memory for the output array jointLocations that can contain at least jointCount of XrHandJointLocationEXT.

The application **must** set jointCount as described by the XrHandJointSetEXT when creating the XrHandTrackerEXT otherwise the runtime **must** return XR_ERROR_VALIDATION_FAILURE.

The runtime **must** return jointLocations representing the range of motion of a human hand, without any obstructions. Input systems that obstruct the movement of the user's hand (eg: a held controller preventing the user from making a fist) or that have only limited ability to track finger positions **must** use the information available to them to emulate an unobstructed range of motion.

The runtime **must** update the jointLocations array ordered so that the application can index elements using the corresponding hand joint enum (e.g. XrHandJointEXT) as described by XrHandJointSetEXT when creating the XrHandTrackerEXT. For example, when the XrHandTrackerEXT is created with XR_HAND_JOINT_SET_DEFAULT_EXT, the application **must** set the jointCount to XR_HAND_JOINT_COUNT_EXT, and the runtime **must** fill the jointLocations array ordered so that it may be indexed by the XrHandJointEXT enum.

If the returned isActive is true, the runtime **must** return all joint locations with both XR_SPACE_LOCATION_POSITION_VALID_BIT and XR_SPACE_LOCATION_ORIENTATION_VALID_BIT set. Although, in this case, some joint space locations **may** be untracked (i.e. XR_SPACE_LOCATION_POSITION_TRACKED_BIT or XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT is unset).

If the returned isActive is false, it indicates the hand tracker did not detect the hand input or the application lost input focus. In this case, the runtime **must** return all jointLocations with neither XR_SPACE_LOCATION_POSITION_VALID_BIT nor XR_SPACE_LOCATION_ORIENTATION_VALID_BIT set.

Valid Usage (Implicit)

- The `XR_EXT_hand_tracking` extension **must** be enabled prior to using XrHandJointLocationsEXT
- `type` **must** be `XR_TYPE_HAND_JOINT_LOCATIONS_EXT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrHandJointVelocitiesEXT
- `jointLocations` **must** be a pointer to an array of `jointCount` XrHandJointLocationEXT structures
- The `jointCount` parameter **must** be greater than `0`

XrHandJointLocationEXT structure describes the position, orientation, and radius of a hand joint.

```
typedef struct XrHandJointLocationEXT {
    XrSpaceLocationFlags    locationFlags;
    XrPosef                 pose;
    float                   radius;
} XrHandJointLocationEXT;
```

Member Descriptions

- `locationFlags` is a bitfield, with bit masks defined in XrSpaceLocationFlagBits, to indicate which members contain valid data. If none of the bits are set, no other fields in this structure **should** be considered to be valid or meaningful.
- `pose` is an XrPosef defining the position and orientation of the origin of a hand joint within the reference frame of the corresponding XrHandJointsLocateInfoEXT::`baseSpace`.
- `radius` is a `float` value radius of the corresponding joint in units of meters.

If the returned `locationFlags` has `XR_SPACE_LOCATION_POSITION_VALID_BIT` set, the returned radius **must** be a positive value.

If the returned `locationFlags` has `XR_SPACE_LOCATION_POSITION_VALID_BIT` unset, the returned radius value is undefined and should be avoided.

Valid Usage (Implicit)

- The XR_EXT_hand_tracking extension **must** be enabled prior to using XrHandJointLocationEXT
- locationFlags **must** be 0 or a valid combination of XrSpaceLocationFlagBits values

The application can chain an XrHandJointVelocitiesEXT structure to the next pointer of XrHandJointLocationsEXT when calling xrLocateHandJointsEXT to retrieve the hand joint velocities.

```
typedef struct XrHandJointVelocitiesEXT {
    XrStructureType            type;
    void*                      next;
    uint32_t                   jointCount;
    XrHandJointVelocityEXT*    jointVelocities;
} XrHandJointVelocitiesEXT;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- jointCount is a uint32_t describing the number of elements in jointVelocities array.
- jointVelocities is an array of XrHandJointVelocityEXT receiving the returned hand joint velocities.

The application **must** allocate the memory for the output array jointVelocities that can contain at least jointCount of XrHandJointVelocityEXT.

The application **must** input jointCount as described by the XrHandJointSetEXT when creating the XrHandTrackerEXT. Otherwise, the runtime **must** return XR_ERROR_VALIDATION_FAILURE.

The runtime **must** update the jointVelocities array in the order so that the application can index elements using the corresponding hand joint enum (e.g. XrHandJointEXT) as described by the XrHandJointSetEXT when creating the XrHandTrackerEXT. For example, when the XrHandTrackerEXT is created with XR_HAND_JOINT_SET_DEFAULT_EXT, the application **must** set the jointCount to XR_HAND_JOINT_COUNT_EXT, and the returned jointVelocities array **must** be ordered to be indexed by enum XrHandJointEXT enum.

If the returned XrHandJointLocationsEXT::isActive is false, it indicates the hand tracker did not detect a hand input or the application lost input focus. In this case, the runtime **must** return all jointVelocities with neither XR_SPACE_VELOCITY_LINEAR_VALID_BIT nor XR_SPACE_VELOCITY_ANGULAR_VALID_BIT set.

If an XrHandJointVelocitiesEXT structure is chained to XrHandJointLocationsEXT::next, the returned XrHandJointLocationsEXT::isActive is true, and the velocity is observed or can be calculated by the runtime, the runtime **must** fill in the linear velocity of each hand joint within the reference frame of baseSpace and set the XR_SPACE_VELOCITY_LINEAR_VALID_BIT. Similarly, if an XrHandJointVelocitiesEXT structure is chained to XrHandJointLocationsEXT::next, the returned XrHandJointLocationsEXT ::isActive is true, and the *angular velocity* is observed or can be calculated by the runtime, the runtime **must** fill in the angular velocity of each joint within the reference frame of baseSpace and set the XR_SPACE_VELOCITY_ANGULAR_VALID_BIT.

Valid Usage (Implicit)

- The XR_EXT_hand_tracking extension **must** be enabled prior to using XrHandJointVelocitiesEXT
- type **must** be XR_TYPE_HAND_JOINT_VELOCITIES_EXT
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- jointVelocities **must** be a pointer to an array of jointCount XrHandJointVelocityEXT structures
- The jointCount parameter **must** be greater than 0

XrHandJointVelocityEXT structure describes the linear and angular velocity of a hand joint.

```
typedef struct XrHandJointVelocityEXT {
    XrSpaceVelocityFlags    velocityFlags;
    XrVector3f              linearVelocity;
    XrVector3f              angularVelocity;
} XrHandJointVelocityEXT;
```

Member Descriptions

- velocityFlags is a bitfield, with bit masks defined in XrSpaceVelocityFlagBits, to indicate which members contain valid data. If none of the bits are set, no other fields in this structure **should** be considered to be valid or meaningful.
- linearVelocity is the relative linear velocity of the hand joint with respect to and expressed in the reference frame of the corresponding XrHandJointsLocateInfoEXT::baseSpace, in units of meters per second.
- angularVelocity is the relative angular velocity of the hand joint with respect to the corresponding XrHandJointsLocateInfoEXT::baseSpace. The vector's direction is expressed in the reference frame of the corresponding XrHandJointsLocateInfoEXT::baseSpace and is parallel to the rotational axis of the hand joint. The vector's magnitude is the relative angular speed of the hand joint in radians per second. The vector follows the right-hand rule for torque/rotation.

Valid Usage (Implicit)

- The XR_EXT_hand_tracking extension **must** be enabled prior to using XrHandJointVelocityEXT
- velocityFlags **must** be a valid combination of XrSpaceVelocityFlagBits values
- velocityFlags **must** not be 0

Example code for locating hand joints

The following example code demonstrates how to locate all hand joints relative to a world space.

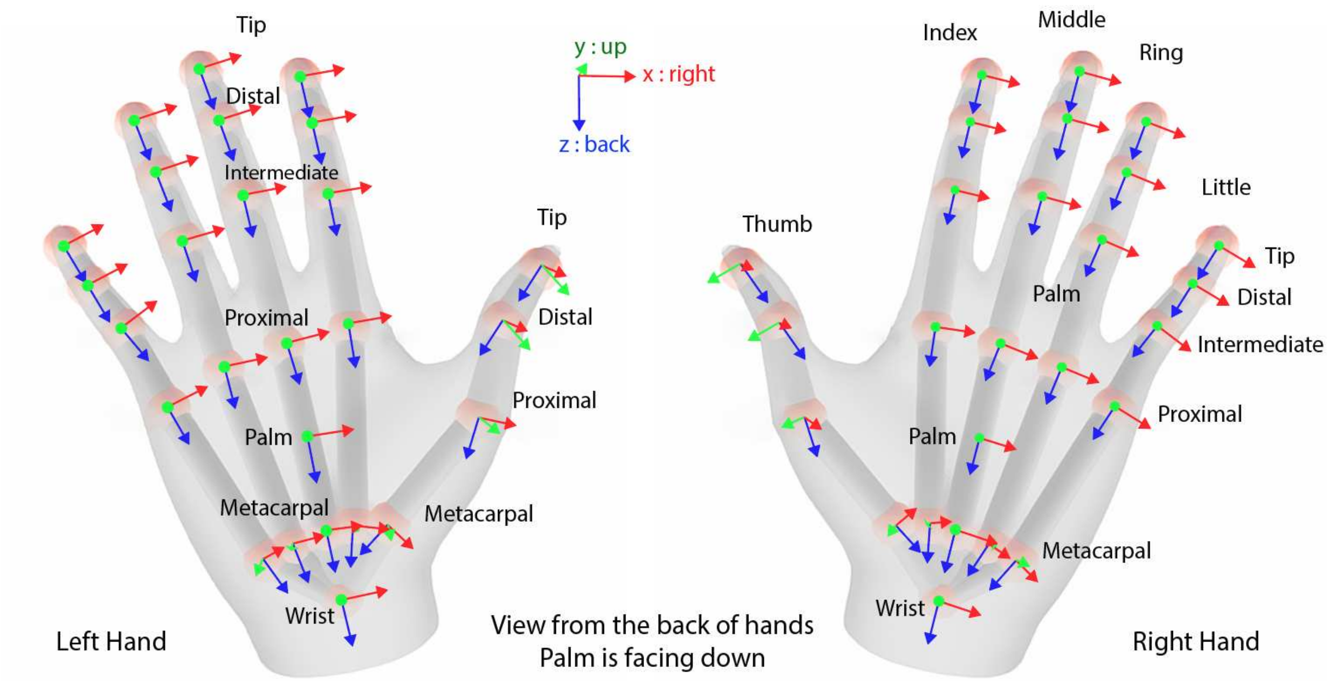
```
XrInstance instance; // previously initialized
XrSystemId systemId; // previously initialized
XrSession session;   // previously initialized
XrSpace worldSpace;  // previously initialized, e.g. from
                     // XR_REFERENCE_SPACE_TYPE_LOCAL

// Inspect hand tracking system properties
XrSystemHandTrackingPropertiesEXT handTrackingSystemProperties{
    XR_TYPE_SYSTEM_HAND_TRACKING_PROPERTIES_EXT};
XrSystemProperties systemProperties{XR_TYPE_SYSTEM_PROPERTIES,
                                    &handTrackingSystemProperties};
CHK_XR(xrGetSystemProperties(instance, systemId, &systemProperties));
if (!handTrackingSystemProperties.supportsHandTracking) {
    // The system does not support hand tracking
    return;
}
```

```
// Get function pointer for xrCreateHandTrackerEXT
PFN_xrCreateHandTrackerEXT pfnCreateHandTrackerEXT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrCreateHandTrackerEXT",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                             &pfnCreateHandTrackerEXT)));

// Create a hand tracker for left hand that tracks default set of hand joints.
XrHandTrackerEXT leftHandTracker{};
{
    XrHandTrackerCreateInfoEXT createInfo{XR_TYPE_HAND_TRACKER_CREATE_INFO_EXT};
    createInfo.hand = XR_HAND_LEFT_EXT;
    createInfo.handJointSet = XR_HAND_JOINT_SET_DEFAULT_EXT;
    CHK_XR(pfnCreateHandTrackerEXT(session, &createInfo, &leftHandTracker));
}

// Allocate buffers to receive joint location and velocity data before frame
// loop starts
XrHandJointLocationEXT jointLocations[XR_HAND_JOINT_COUNT_EXT];
XrHandJointVelocityEXT jointVelocities[XR_HAND_JOINT_COUNT_EXT];

XrHandJointVelocitiesEXT velocities{XR_TYPE_HAND_JOINT_VELOCITIES_EXT};
velocities.jointCount = XR_HAND_JOINT_COUNT_EXT;
velocities.jointVelocities = jointVelocities;

XrHandJointLocationsEXT locations{XR_TYPE_HAND_JOINT_LOCATIONS_EXT};
locations.next = &velocities;
locations.jointCount = XR_HAND_JOINT_COUNT_EXT;
locations.jointLocations = jointLocations;

// Get function pointer for xrLocateHandJointsEXT
PFN_xrLocateHandJointsEXT pfnLocateHandJointsEXT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrLocateHandJointsEXT",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                             &pfnLocateHandJointsEXT)));
while (1) {
    // ...
    // For every frame in frame loop
    // ...
    XrFrameState frameState; // previously returned from xrWaitFrame
    const XrTime time = frameState.predictedDisplayTime;

    XrHandJointsLocateInfoEXT locateInfo{XR_TYPE_HAND_JOINTS_LOCATE_INFO_EXT};
    locateInfo.baseSpace = worldSpace;
    locateInfo.time = time;

    CHK_XR(pfnLocateHandJointsEXT(leftHandTracker, &locateInfo, &locations));

    if (locations.isActive) {
```

```
        // The returned joint location array can be directly indexed with
        // XrHandJointEXT enum.
        const XrPosef &indexTipInWorld =
            jointLocations[XR_HAND_JOINT_INDEX_TIP_EXT].pose;
        const XrPosef &thumbTipInWorld =
            jointLocations[XR_HAND_JOINT_THUMB_TIP_EXT].pose;

        // using the returned radius and velocity of index finger tip.
        const float indexTipRadius =
            jointLocations[XR_HAND_JOINT_INDEX_TIP_EXT].radius;
        const XrHandJointVelocityEXT &indexTipVelocity =
            jointVelocities[XR_HAND_JOINT_INDEX_TIP_EXT];
    }
}
```

Conventions of hand joints

This extension defines 26 joints for hand tracking: 4 joints for the thumb finger, 5 joints for the other four fingers, and the wrist and palm of the hands.



```
typedef enum XrHandJointEXT {
    XR_HAND_JOINT_PALM_EXT = 0,
    XR_HAND_JOINT_WRIST_EXT = 1,
    XR_HAND_JOINT_THUMB_METACARPAL_EXT = 2,
    XR_HAND_JOINT_THUMB_PROXIMAL_EXT = 3,
    XR_HAND_JOINT_THUMB_DISTAL_EXT = 4,
    XR_HAND_JOINT_THUMB_TIP_EXT = 5,
    XR_HAND_JOINT_INDEX_METACARPAL_EXT = 6,
    XR_HAND_JOINT_INDEX_PROXIMAL_EXT = 7,
    XR_HAND_JOINT_INDEX_INTERMEDIATE_EXT = 8,
    XR_HAND_JOINT_INDEX_DISTAL_EXT = 9,
    XR_HAND_JOINT_INDEX_TIP_EXT = 10,
    XR_HAND_JOINT_MIDDLE_METACARPAL_EXT = 11,
    XR_HAND_JOINT_MIDDLE_PROXIMAL_EXT = 12,
    XR_HAND_JOINT_MIDDLE_INTERMEDIATE_EXT = 13,
    XR_HAND_JOINT_MIDDLE_DISTAL_EXT = 14,
    XR_HAND_JOINT_MIDDLE_TIP_EXT = 15,
    XR_HAND_JOINT_RING_METACARPAL_EXT = 16,
    XR_HAND_JOINT_RING_PROXIMAL_EXT = 17,
    XR_HAND_JOINT_RING_INTERMEDIATE_EXT = 18,
    XR_HAND_JOINT_RING_DISTAL_EXT = 19,
    XR_HAND_JOINT_RING_TIP_EXT = 20,
    XR_HAND_JOINT_LITTLE_METACARPAL_EXT = 21,
    XR_HAND_JOINT_LITTLE_PROXIMAL_EXT = 22,
    XR_HAND_JOINT_LITTLE_INTERMEDIATE_EXT = 23,
    XR_HAND_JOINT_LITTLE_DISTAL_EXT = 24,
    XR_HAND_JOINT_LITTLE_TIP_EXT = 25,
    XR_HAND_JOINT_MAX_ENUM_EXT = 0x7FFFFFFF
} XrHandJointEXT;
```

The finger joints, except the tips, are named after the corresponding bone at the further end of the bone from the finger tips. The joint's orientation is defined at a fully opened hand pose facing down as in the above picture.

> *Note*
>
> Many applications and game engines use names to identify joints rather than using indices. If possible, applications should use the joint name part of the XrHandJointEXT enum plus a hand identifier to help prevent joint name clashes (e.g. Index_Metacarpal_L, Thumb_Tip_R). Using consistent names increases the portability of assets between applications and engines. Including the hand in the identifier prevents ambiguity when both hands are used in the same skeleton, such as when they are combined with additional joints to form a full body skeleton.

The backward (+Z) direction is parallel to the corresponding bone and points away from the finger tip. The up (+Y) direction is pointing out of the back of and perpendicular to the corresponding finger nail at the fully opened hand pose. The X direction is perpendicular to Y and Z and follows the right hand rule.

The wrist joint is located at the pivot point of the wrist which is location invariant when twisting hand without moving the forearm. The backward (+Z) direction is parallel to the line from wrist joint to middle finger metacarpal joint, and points away from the finger tips. The up (+Y) direction points out towards back of hand and perpendicular to the skin at wrist. The X direction is perpendicular to the Y and Z directions and follows the right hand rule.

The palm joint is located at the center of the middle finger's metacarpal bone. The backward (+Z) direction is parallel to the middle finger's metacarpal bone, and points away from the finger tips. The up (+Y) direction is perpendicular to palm surface and pointing towards the back of the hand. The X direction is perpendicular to the Y and Z directions and follows the right hand rule.

The radius of each joint is the distance from the joint to the skin in meters. The application can use a sphere at the joint location with joint radius for collision detection for interactions, such as pushing a virtual button using the index finger tip.

For example, suppose the radius of the palm joint is `r` then the app **can** offset `{0, -r, 0}` to palm joint location to get the surface of hand palm center, or offset `{0, r, 0}` to get the back surface of the hand.

Note that the palm joint for the hand tracking is not the same as *…/input/grip/pose* when hand tracking is provided by controller tracking. A "grip" pose is located at the center of the controller handle when user is holding a controller, outside of the user's hand. A "palm" pose is located at the center of middle finger metacarpal bone which is inside the user's hand.

```
#define XR_HAND_JOINT_COUNT_EXT 26
```

XR_HAND_JOINT_COUNT_EXT defines the number of hand joint enumerants defined in XrHandJointEXT

New Object Types

- XrHandTrackerEXT

New Flag Types

New Enum Constants

- XR_HAND_JOINT_COUNT_EXT

XrObjectType enumeration is extended with:

- `XR_OBJECT_TYPE_HAND_TRACKER_EXT`

XrStructureType enumeration is extended with:

- `XR_TYPE_SYSTEM_HAND_TRACKING_PROPERTIES_EXT`
- `XR_TYPE_HAND_TRACKER_CREATE_INFO_EXT`
- `XR_TYPE_HAND_JOINTS_LOCATE_INFO_EXT`
- `XR_TYPE_HAND_JOINT_LOCATIONS_EXT`
- `XR_TYPE_HAND_JOINT_VELOCITIES_EXT`

New Enums

- XrHandEXT
- XrHandJointEXT
- XrHandJointSetEXT

New Structures

- XrSystemHandTrackingPropertiesEXT
- XrHandTrackerCreateInfoEXT
- XrHandJointsLocateInfoEXT
- XrHandJointLocationEXT
- XrHandJointVelocityEXT
- XrHandJointLocationsEXT
- XrHandJointVelocitiesEXT

New Functions

- xrCreateHandTrackerEXT
- xrDestroyHandTrackerEXT
- xrLocateHandJointsEXT

Issues

Version History

- Revision 1, 2019-09-16 (Yin LI)
  - Initial extension description
- Revision 2, 2020-04-20 (Yin LI)
  - Replace hand joint spaces to locate hand joints function.
- Revision 3, 2021-04-13 (Ryan Pavlik, Rune Berg)
  - Fix example code to properly use `xrGetInstanceProcAddr`.

    - Add recommended bone names
- Revision 4, 2021-04-15 (Rune Berg)
    - Clarify that use of this extension produces an unobstructed hand range of motion.

12.28. XR_EXT_hp_mixed_reality_controller

Name String

`XR_EXT_hp_mixed_reality_controller`

Extension Type

Instance extension

Registered Extension Number

96

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2020-06-08

IP Status

No known IP claims.

Contributors

Alain Zanchetta, Microsoft
Lachlan Ford, Microsoft
Alex Turner, Microsoft
Yin Li, Microsoft
Nathan Nuber, HP Inc.

Overview

This extension added a new interaction profile path for the HP Reverb G2 Controllers:

- */interaction_profiles/hp/mixed_reality_controller*

Valid for the user paths

- */user/hand/left*
- */user/hand/right*

Supported component paths:

- On /user/hand/left only
  - *…/input/x/click*
  - *…/input/y/click*
- On /user/hand/right only
  - *…/input/a/click*
  - *…/input/b/click*
- On both hands
  - *…/input/menu/click*
  - *…/input/squeeze/value*
  - *…/input/trigger/value*
  - *…/input/thumbstick/x*
  - *…/input/thumbstick/y*
  - *…/input/thumbstick/click*
  - *…/input/grip/pose*
  - *…/input/aim/pose*
  - *…/output/haptic*

Version History

- Revision 1, 2020-06-08 (Yin Li)
  - Initial extension proposal

12.29. XR_EXT_performance_settings

Name String

`XR_EXT_performance_settings`

Extension Type

Instance extension

Registered Extension Number

16

Revision

3

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2021-04-14

IP Status

No known IP claims.

Contributors

Armelle Laine, Qualcomm Technologies Inc, on behalf of Qualcomm Innovation Center, Inc
Ryan Pavlik, Collabora

12.29.1. Overview

This extension defines an API for the application to give performance hints to the runtime and for the runtime to send performance related notifications back to the application. This allows both sides to dial in a suitable compromise between needed CPU and GPU performance, thermal sustainability and a consistent good user experience throughout the session.

The goal is to render frames consistently, in time, under varying system load without consuming more energy than necessary.

In summary, the APIs allow:

- setting performance level hints
- receiving performance related notifications

12.29.2. Setting Performance Levels Hints

Performance level hint definition

The XR performance level hints for a given hardware system are expressed as a level XrPerfSettingsLevelEXT for each of the XR-critical processing domains XrPerfSettingsDomainEXT (currently defined is a CPU and a GPU domain):

```
typedef enum XrPerfSettingsDomainEXT {
    XR_PERF_SETTINGS_DOMAIN_CPU_EXT = 1,
    XR_PERF_SETTINGS_DOMAIN_GPU_EXT = 2,
    XR_PERF_SETTINGS_DOMAIN_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsDomainEXT;
```

```
typedef enum XrPerfSettingsLevelEXT {
    XR_PERF_SETTINGS_LEVEL_POWER_SAVINGS_EXT = 0,
    XR_PERF_SETTINGS_LEVEL_SUSTAINED_LOW_EXT = 25,
    XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT = 50,
    XR_PERF_SETTINGS_LEVEL_BOOST_EXT = 75,
    XR_PERF_SETTINGS_LEVEL_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsLevelEXT;
```

This extension defines platform-independent level hints:

- XR_PERF_SETTINGS_LEVEL_POWER_SAVINGS_EXT is used by the application to indicate that it enters a non-XR section (head-locked / static screen), during which power savings are to be prioritized. Consistent XR compositing, consistent frame rendering, and low latency are not needed.
- XR_PERF_SETTINGS_LEVEL_SUSTAINED_LOW_EXT is used by the application to indicate that it enters a low and stable complexity section, during which reducing power is more important than occasional late rendering frames. With such a hint, the XR Runtime still strives for consistent XR compositing (no tearing) within a thermally sustainable range(*), but is allowed to take measures to reduce power, such as increasing latencies or reducing headroom.
- XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT is used by the application to indicate that it enters a high or dynamic complexity section, during which the XR Runtime strives for consistent XR compositing and frame rendering within a thermally sustainable range(*).
- XR_PERF_SETTINGS_LEVEL_BOOST_EXT is used to indicate that the application enters a section with very high complexity, during which the XR Runtime is allowed to step up beyond the thermally sustainable range. As not thermally sustainable, this level is meant to be used for short-term durations (< 30 seconds).

(*) If the application chooses one of the two sustainable levels (XR_PERF_SETTINGS_LEVEL_SUSTAINED_LOW_EXT or XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT), the device may still run into thermal limits under non-nominal circumstances (high room temperature, additional background loads, extended device operation) and therefore the application should also in the sustainable modes be prepared to react to performance notifications (in particular XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT and XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT in the thermal sub-domain, see Notification level definition).

The XR Runtime shall select XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT as the default hint if the application does not provide any. The function to call for setting performance level hints is xrPerfSettingsSetPerformanceLevelEXT.

```
XrResult xrPerfSettingsSetPerformanceLevelEXT(
    XrSession                                   session,
    XrPerfSettingsDomainEXT                     domain,
    XrPerfSettingsLevelEXT                      level);
```

Example of using the short-term boost level hint

For a limited amount of time, both the Mobile and PC systems can provide a higher level of performance than is thermally sustainable. It is desirable to make this extra computational power available for short complex scenes, then go back to a sustainable lower level. This section describes means for the application developer to apply settings directing the runtime to boost performance for a short-term duration.

The application developer must pay attention to keep these boost periods very short and carefully monitor the side effects, which may vary a lot between different hardware systems.

Sample code for temporarily boosting the performance

```
extern XrInstance instance; ①
extern XrSession session;

// Get function pointer for xrPerfSettingsSetPerformanceLevelEXT
PFN_xrPerfSettingsSetPerformanceLevelEXT pfnPerfSettingsSetPerformanceLevelEXT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrPerfSettingsSetPerformanceLevelEXT",
                             (PFN_xrVoidFunction*)(
                             &pfnPerfSettingsSetPerformanceLevelEXT)));

// before entering the high complexity section
pfnPerfSettingsSetPerformanceLevelEXT(session, XR_PERF_SETTINGS_DOMAIN_CPU_EXT,
XR_PERF_SETTINGS_LEVEL_BOOST_EXT); ②
pfnPerfSettingsSetPerformanceLevelEXT(session, XR_PERF_SETTINGS_DOMAIN_GPU_EXT,
XR_PERF_SETTINGS_LEVEL_BOOST_EXT);

// entering the high complexity section
// ... running
// end of the high complexity section

pfnPerfSettingsSetPerformanceLevelEXT(session, XR_PERF_SETTINGS_DOMAIN_CPU_EXT,
XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT); ③
pfnPerfSettingsSetPerformanceLevelEXT(session, XR_PERF_SETTINGS_DOMAIN_GPU_EXT,
XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT);
```

① we assume that `instance` and `session` are initialized and their handles are available

② setting performance level to `XR_PERF_SETTINGS_LEVEL_BOOST_EXT` on both CPU and GPU domains

③ going back to the sustainable `XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT`

Example of using the sustained low level hint for the CPU domain

power reduction sample code

```
extern XrInstance instance; ①
extern XrSession session;

// Get function pointer for xrPerfSettingsSetPerformanceLevelEXT
PFN_xrPerfSettingsSetPerformanceLevelEXT pfnPerfSettingsSetPerformanceLevelEXT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrPerfSettingsSetPerformanceLevelEXT",
                             (PFN_xrVoidFunction*)(
                             &pfnPerfSettingsSetPerformanceLevelEXT)));

// before entering a low CPU complexity section
pfnPerfSettingsSetPerformanceLevelEXT(session, XR_PERF_SETTINGS_DOMAIN_CPU_EXT,
XR_PERF_SETTINGS_LEVEL_SUSTAINED_LOW_EXT);
pfnPerfSettingsSetPerformanceLevelEXT(session, XR_PERF_SETTINGS_DOMAIN_GPU_EXT,
XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT); ②

// entering the low complexity section
// ... running
// end of the low complexity section

pfnPerfSettingsSetPerformanceLevelEXT(session, XR_PERF_SETTINGS_DOMAIN_CPU_EXT,
XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT); ③
```

① we assume that `instance` and `session` are initialized and their handles are available

② the developer may choose to only reduce CPU domain and keep the GPU domain at `XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT`

③ going back to the sustainable `XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT` for CPU

12.29.3. Receiving Performance Related Notifications

The XR runtime shall provide performance related notifications to the application in the following situations:

- the compositing performance within the runtime has reached a new level, either improved or degraded from the previous one (`subDomain` is set to `XR_PERF_SETTINGS_SUB_DOMAIN_COMPOSITING_EXT`)
- the application rendering performance has reached a new level, either improved or degraded from the previous one (`subDomain` is set to `XR_PERF_SETTINGS_SUB_DOMAIN_RENDERING_EXT`)
- the temperature of the device has reached a new level, either improved or degraded from the previous one (`subDomain` is set to `XR_PERF_SETTINGS_SUB_DOMAIN_THERMAL_EXT`).

When degradation is observed, the application **should** take measures reducing its workload, helping the compositing or rendering `subDomain` to meet their deadlines, or the thermal `subDomain` to avoid or stop throttling. When improvement is observed, the application can potentially rollback some of its mitigations.

```
typedef struct XrEventDataPerfSettingsEXT {
    XrStructureType                        type;
    const void*                            next;
    XrPerfSettingsDomainEXT                domain;
    XrPerfSettingsSubDomainEXT             subDomain;
    XrPerfSettingsNotificationLevelEXT     fromLevel;
    XrPerfSettingsNotificationLevelEXT     toLevel;
} XrEventDataPerfSettingsEXT;
```

```
typedef enum XrPerfSettingsSubDomainEXT {
    XR_PERF_SETTINGS_SUB_DOMAIN_COMPOSITING_EXT = 1,
    XR_PERF_SETTINGS_SUB_DOMAIN_RENDERING_EXT = 2,
    XR_PERF_SETTINGS_SUB_DOMAIN_THERMAL_EXT = 3,
    XR_PERF_SETTINGS_SUB_DOMAIN_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsSubDomainEXT;
```

Compositing Sub-Domain

One of the major functions the runtime shall provide is the timely compositing of the submitted layers in the background. The runtime has to share the CPU and GPU system resources for this operation with the application. Since this is extremely time sensitive - the head room is only a few milliseconds - the runtime may have to ask the application via notifications to cooperate and relinquish some usage of the indicated resource (CPU or GPU domain). Performance issues in this area that the runtime notices are notified to the application with the `subDomain` set to `XR_PERF_SETTINGS_SUB_DOMAIN_COMPOSITING_EXT`.

Rendering Sub-Domain

The application submits rendered layers to the runtime for compositing. Performance issues in this area that the runtime notices (i.e. missing submission deadlines) are notified to the application with the `subDomain` set to `XR_PERF_SETTINGS_SUB_DOMAIN_RENDERING_EXT`.

Thermal Sub-Domain

XR applications run at a high-performance level during long periods of time, across a game or an entire movie session. As form factors shrink, especially on mobile solutions, the risk of reaching die thermal runaway or reaching the limits on skin and battery temperatures increases. When thermal limits are reached, the device mitigates the heat generation leading to severe performance reductions, which greatly affects user experience (dropped frames, high latency).

Better than dropping frames when it is too late, pro-active measures from the application should be encouraged.

The performance notification with the `subDomain` set to `XR_PERF_SETTINGS_SUB_DOMAIN_THERMAL_EXT` provides an early warning allowing the application to take mitigation actions.

Notification level definition

The levels are defined as follows:

```
typedef enum XrPerfSettingsNotificationLevelEXT {
    XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT = 0,
    XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT = 25,
    XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT = 75,
    XR_PERF_SETTINGS_NOTIFICATION_LEVEL_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsNotificationLevelEXT;
```

- `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT` notifies that the sub-domain has reached a level where no further actions other than currently applied are necessary.
- `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT` notifies that the sub-domain has reached an early warning level where the application should start proactive mitigation actions with the goal to return to the `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT` level.
- `XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT` notifies that the sub-domain has reached a critical level with significant performance degradation. The application should take drastic mitigation action.

The above definitions summarize the broad interpretation of the notification levels, however sub-domain specific definitions of each level and their transitions are specified below:

- `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT`
  - For the compositing sub-domain, `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT` indicates that the composition headroom is consistently being met with sufficient margin.
    Getting into `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT` from `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT` indicates that the composition headroom was consistently **met with sufficient margin during a sufficient time period**.
  - For the rendering sub-domain, `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT` indicates that frames are being submitted in time to be used by the compositor.
    Getting into `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT` from `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT` indicates that **during a sufficient time period**, **none** of the due layers was **too late** to be picked up by the compositor.
  - For the thermal sub-domain, `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT` indicates that the current load should be sustainable in the near future.
    Getting into `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT` from `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT` indicates that the runtime does not presuppose any further temperature mitigation action on the application side, other than the current ones.
- `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT`
  - For the compositing sub-domain, `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT` indicates that the compositing headroom of the current frame was met but the margin is considered insufficient by the runtime, and the application **should** reduce its workload in the notified domain to solve

this problem.
Getting into XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT from XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT indicates that the compositing deadline was **not missed during a sufficient time period**.

  - For the rendering sub-domain, XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT indicates that at least one layer is regularly late to be picked up by the compositor, resulting in a degraded user experience, and that the application should take action to consistently provide frames in a more timely manner.
Getting into XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT from XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT indicates that **the runtime has stopped any of its own independent actions** which are tied to the XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT level.

  - For the thermal sub-domain, the XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT indicates that the runtime expects the device to overheat under the current load, and that the application should take mitigating action in order to prevent thermal throttling.
Getting into XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT from XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT indicates that **the underlying system thermal throttling has stopped**.

- XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT
  - For the compositing sub-domain, XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT indicates that composition can no longer be maintained under the current workload. The runtime may take independent action that will interfere with the application (e.g. limiting the framerate, ignoring submitted layers, or shutting down the application) in order to correct this problem.
  - For the rendering sub-domain, XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT indicates that at least one layer is **too often** late to be picked up by the compositor, and consequently the runtime may take independent action that will interfere with the application (e.g. informing the user that the application is not responding, displaying a tracking environment in order to maintain user orientation).
  - For the thermal sub-domain, XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT indicates that the **underlying system is taking measures, such as thermal throttling** to reduce the temperature, impacting the XR experience.

Leaving XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT indicates that any mitigating actions by the runtime (e.g. down-clocking the device to stay within thermal limits) have ended.

Performance Settings API Reference

xrPerfSettingsSetPerformanceLevelEXT

```
XrResult xrPerfSettingsSetPerformanceLevelEXT(
    XrSession                                   session,
    XrPerfSettingsDomainEXT                     domain,
    XrPerfSettingsLevelEXT                      level);
```

Parameter Descriptions

- session is a valid XrSession handle.
- domain: the processing domain for which the level hint is applied
- level: the level hint to be applied

Valid Usage (Implicit)

- The XR_EXT_performance_settings extension **must** be enabled prior to calling xrPerfSettingsSetPerformanceLevelEXT
- session **must** be a valid XrSession handle
- domain **must** be a valid XrPerfSettingsDomainEXT value
- level **must** be a valid XrPerfSettingsLevelEXT value

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_FUNCTION_UNSUPPORTED

Refer to Performance level hint definition for the definition of the level enumerations.

XrEventDataPerformanceSettingsEXT

```
typedef struct XrEventDataPerfSettingsEXT {
    XrStructureType                       type;
    const void*                           next;
    XrPerfSettingsDomainEXT               domain;
    XrPerfSettingsSubDomainEXT            subDomain;
    XrPerfSettingsNotificationLevelEXT    fromLevel;
    XrPerfSettingsNotificationLevelEXT    toLevel;
} XrEventDataPerfSettingsEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `domain` : processing domain in which a threshold has been crossed
- `subDomain` : system area in which a threshold has been crossed
- `fromLevel` : enumerated notification level which has been exited
- `toLevel` : enumerated notification level which has been entered

Valid Usage (Implicit)

- The `XR_EXT_performance_settings` extension **must** be enabled prior to using XrEventDataPerfSettingsEXT
- `type` **must** be `XR_TYPE_EVENT_DATA_PERF_SETTINGS_EXT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `domain` **must** be a valid XrPerfSettingsDomainEXT value
- `subDomain` **must** be a valid XrPerfSettingsSubDomainEXT value
- `fromLevel` **must** be a valid XrPerfSettingsNotificationLevelEXT value
- `toLevel` **must** be a valid XrPerfSettingsNotificationLevelEXT value

```
typedef enum XrPerfSettingsDomainEXT {
    XR_PERF_SETTINGS_DOMAIN_CPU_EXT = 1,
    XR_PERF_SETTINGS_DOMAIN_GPU_EXT = 2,
    XR_PERF_SETTINGS_DOMAIN_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsDomainEXT;
```

```
typedef enum XrPerfSettingsSubDomainEXT {
    XR_PERF_SETTINGS_SUB_DOMAIN_COMPOSITING_EXT = 1,
    XR_PERF_SETTINGS_SUB_DOMAIN_RENDERING_EXT = 2,
    XR_PERF_SETTINGS_SUB_DOMAIN_THERMAL_EXT = 3,
    XR_PERF_SETTINGS_SUB_DOMAIN_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsSubDomainEXT;
```

```
typedef enum XrPerfSettingsNotificationLevelEXT {
    XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT = 0,
    XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT = 25,
    XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT = 75,
    XR_PERF_SETTINGS_NOTIFICATION_LEVEL_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsNotificationLevelEXT;
```

Version History

- Revision 1, 2017-11-30 (Armelle Laine)
- Revision 2, 2021-04-13 (Ryan Pavlik)
  - Correctly show function pointer retrieval in sample code
  - Fix sample code callouts
- Revision 3, 2021-04-14 (Ryan Pavlik)
  - Fix missing error code

12.30. XR_EXT_samsung_odyssey_controller

Name String

`XR_EXT_samsung_odyssey_controller`

Extension Type

Instance extension

Registered Extension Number

95

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2020-06-08

IP Status

No known IP claims.

Contributors

Lachlan Ford, Microsoft
Alex Turner, Microsoft
Yin Li, Microsoft
Philippe Harscoet, Samsung

Overview

This extension enables the application to differentiate the newer form factor of motion controller released with the Samsung Odyssey headset. It enables the application to customize the appearance and experience of the controller differently from the original mixed reality motion controller.

This extension added a new interaction profile */interaction_profiles/samsung/odyssey_controller* to describe the Odyssey controller. The action bindings of this interaction profile work exactly the same as the */interaction_profiles/microsoft/motion_controller* in terms of valid user paths and supported input and output component paths.

If the application doesn't do its own custom rendering for specific motion controllers, it **should** avoid using this extension and instead just use *…/microsoft/motion_controller*, as runtimes **should** treat both controllers equally when applications declare action bindings only for that profile.

If the application wants to customize rendering for specific motion controllers, it **should** setup the suggested bindings for *…/samsung/odyssey_controller* the same as *…/microsoft/motion_controller* when calling xrSuggestInteractionProfileBindings, and expect the same action bindings. Then the application **can** listen to the XrEventDataInteractionProfileChanged event and inspect the returned interaction profile from xrGetCurrentInteractionProfile to differentiate which controller is being used by the user, and hence customize the appearance or experience of the motion controller specifically for the form factor of *…/samsung/odyssey_controller.*

Version History

- Revision 1, 2020-06-08 (Yin Li)
  - Initial extension proposal

12.31. XR_EXT_thermal_query

Name String

`XR_EXT_thermal_query`

Extension Type

Instance extension

Registered Extension Number

17

Revision

2

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2021-04-14

IP Status

No known IP claims.

Contributors

Armelle Laine, Qualcomm Technologies Inc, on behalf of Qualcomm Innovation Center, Inc

12.31.1. Overview

This extension provides an API to query a domain's current thermal warning level and current thermal trend.

12.31.2. Querying the current thermal level and trend

This query allows to determine the extent and urgency of the needed workload reduction and to verify that the mitigation measures efficiently reduce the temperature.
This query allows the application to retrieve the current `notificationLevel`, allowing to quickly verify whether the underlying system's thermal throttling is still in effect.
It also provides the application with the remaining temperature headroom (`tempHeadroom`) until thermal throttling occurs, and the current rate of change (`tempSlope`).
The most critical temperature of the domain is the one which is currently most likely to be relevant for thermal throttling.

To query the status of a given domain:

```
XrResult xrThermalGetTemperatureTrendEXT(
    XrSession                                   session,
    XrPerfSettingsDomainEXT                     domain,
    XrPerfSettingsNotificationLevelEXT*         notificationLevel,
    float*                                      tempHeadroom,
    float*                                      tempSlope);
```

```
typedef enum XrPerfSettingsDomainEXT {
    XR_PERF_SETTINGS_DOMAIN_CPU_EXT = 1,
    XR_PERF_SETTINGS_DOMAIN_GPU_EXT = 2,
    XR_PERF_SETTINGS_DOMAIN_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsDomainEXT;
```

```
typedef enum XrPerfSettingsNotificationLevelEXT {
    XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT = 0,
    XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT = 25,
    XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT = 75,
    XR_PERF_SETTINGS_NOTIFICATION_LEVEL_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsNotificationLevelEXT;
```

For the definition of the notification levels, see Notification level definition

Thermal Query API Reference

xrThermalGetTemperatureTrendEXT

```
XrResult xrThermalGetTemperatureTrendEXT(
    XrSession                                   session,
    XrPerfSettingsDomainEXT                     domain,
    XrPerfSettingsNotificationLevelEXT*         notificationLevel,
    float*                                      tempHeadroom,
    float*                                      tempSlope);
```

Allows to query the current temperature warning level of a domain, the remaining headroom and the trend.

Parameter Descriptions

- `session` is a valid XrSession handle.
- `domain` : the processing domain
- `notificationLevel` : the current warning level
- `tempHeadroom` : temperature headroom in degrees Celsius, expressing how far the most-critical temperature of the domain is from its thermal throttling threshold temperature.
- `tempSlope` : the current trend in degrees Celsius per second of the most critical temperature of the domain.

Valid Usage (Implicit)

- The `XR_EXT_thermal_query` extension **must** be enabled prior to calling xrThermalGetTemperatureTrendEXT
- `session` **must** be a valid XrSession handle
- `domain` **must** be a valid XrPerfSettingsDomainEXT value
- `notificationLevel` **must** be a pointer to an XrPerfSettingsNotificationLevelEXT value
- `tempHeadroom` **must** be a pointer to a `float` value
- `tempSlope` **must** be a pointer to a `float` value

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_FUNCTION_UNSUPPORTED`

```
typedef enum XrPerfSettingsDomainEXT {
    XR_PERF_SETTINGS_DOMAIN_CPU_EXT = 1,
    XR_PERF_SETTINGS_DOMAIN_GPU_EXT = 2,
    XR_PERF_SETTINGS_DOMAIN_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsDomainEXT;
```

```
typedef enum XrPerfSettingsNotificationLevelEXT {
    XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT = 0,
    XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT = 25,
    XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT = 75,
    XR_PERF_SETTINGS_NOTIFICATION_LEVEL_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsNotificationLevelEXT;
```

Version History

- Revision 1, 2017-11-30 (Armelle Laine)
- Revision 2, 2021-04-14 (Ryan Pavlik, Collabora)
  - Fix missing error code

12.32. XR_EXT_view_configuration_depth_range

Name String

`XR_EXT_view_configuration_depth_range`

Extension Type

Instance extension

Registered Extension Number

47

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2019-08-16

IP Status

No known IP claims.

Contributors

Blake Taylor, Magic Leap
Gilles Cadet, Magic Leap
Michael Liebenow, Magic Leap
Supreet Suresh, Magic Leap
Alex Turner, Microsoft
Bryce Hutchings, Microsoft
Yin Li, Microsoft

Overview

For XR systems there may exist a per view recommended min/max depth range at which content should be rendered into the virtual world. The depth range may be driven by several factors, including user comfort, or fundamental capabilities of the system.

Displaying rendered content outside the recommended min/max depth range would violate the system requirements for a properly integrated application, and can result in a poor user experience due to observed visual artifacts, visual discomfort, or fatigue. The near/far depth values will fall in the range of (0..+infinity] where max(`recommendedNearZ`, `minNearZ`) < min(`recommendedFarZ`, `maxFarZ`). Infinity is defined matching the standard library definition such that std::isinf will return true for a returned infinite value.

In order to provide the application with the appropriate depth range at which to render content for each XrViewConfigurationView, this extension provides additional view configuration information, as defined by XrViewConfigurationDepthRangeEXT, to inform the application of the min/max recommended and absolute distances at which content should be rendered for that view.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_VIEW_CONFIGURATION_DEPTH_RANGE_EXT`

New Enums

New Structures

The XrViewConfigurationDepthRangeEXT structure is defined as:

```
typedef struct XrViewConfigurationDepthRangeEXT {
    XrStructureType    type;
    void*              next;
    float              recommendedNearZ;
    float              minNearZ;
    float              recommendedFarZ;
    float              maxFarZ;
} XrViewConfigurationDepthRangeEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `recommendedNearZ` is the recommended minimum positive distance in meters that content should be rendered for the view to achieve the best user experience.
- `minNearZ` is the absolute minimum positive distance in meters that content should be rendered for the view.
- `recommendedFarZ` is the recommended maximum positive distance in meters that content should be rendered for the view to achieve the best user experience.
- `maxFarZ` is the absolute maximum positive distance in meters that content should be rendered for the view.

When enumerating the view configurations with xrEnumerateViewConfigurationViews, the application **can** provide a pointer to an XrViewConfigurationDepthRangeEXT in the `next` chain of XrViewConfigurationView.

Valid Usage (Implicit)

- The `XR_EXT_view_configuration_depth_range` extension **must** be enabled prior to using XrViewConfigurationDepthRangeEXT
- `type` **must** be `XR_TYPE_VIEW_CONFIGURATION_DEPTH_RANGE_EXT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

New Functions

Issues

Version History

- Revision 1, 2019-10-01 (Blake Taylor)
  - Initial proposal.

12.33. XR_EXT_win32_appcontainer_compatible

Name String

`XR_EXT_win32_appcontainer_compatible`

Extension Type

Instance extension

Registered Extension Number

58

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2019-12-16

IP Status

No known IP claims.

Contributors

Yin Li, Microsoft
Alex Turner, Microsoft
Lachlan Ford, Microsoft

Overview

To minimize opportunities for malicious manipulation, a common practice on the Windows OS is to isolate the application process in an [AppContainer execution environment] (https://docs.microsoft.com/en-us/windows/win32/secauthz/appcontainer-isolation). In order for a runtime to work properly in such an application process, the runtime **must** properly [set ACL to device resources and cross process resources] (https://docs.microsoft.com/en-us/windows/win32/secauthz/implementing-an-appcontainer).

An application running in an AppContainer process **can** request for a runtime to enable such AppContainer compatibility by adding `XR_EXT_WIN32_APPCONTAINER_COMPATIBLE_EXTENSION_NAME` to `enabledExtensionNames` of XrInstanceCreateInfo when calling xrCreateInstance. If the runtime is not capable of running properly within the AppContainer execution environment, it **must** return `XR_ERROR_EXTENSION_NOT_PRESENT`.

If the runtime supports this extension, it **can** further inspect the capability based on the connected device. If the XR system cannot support an AppContainer execution environment, the runtime **must** return `XR_ERROR_FORM_FACTOR_UNAVAILABLE` when the application calls xrGetSystem.

If the call to xrGetSystem successfully returned with a valid `XrSystemId`, the application **can** rely on the runtime working properly in the AppContainer execution environment.

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

New Functions

Issues

Version History

- Revision 1, 2019-12-16 (Yin Li)
  - Initial proposal.

12.34. XR_EPIC_view_configuration_fov

Name String

`XR_EPIC_view_configuration_fov`

Extension Type

Instance extension

Registered Extension Number

60

Revision

2

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2020-03-05

IP Status

No known IP claims.

Contributors

Jules Blok, Epic Games

Overview

This extension allows the application to retrieve the recommended and maximum field-of-view using xrEnumerateViewConfigurationViews. These field-of-view parameters can be used during initialization of the application before creating a session.

The field-of-view given here **should** not be used for rendering, see xrLocateViews to retrieve the field-of-view for rendering.

For views with `fovMutable` set to `XR_TRUE` the maximum field-of-view **should** specify the upper limit that runtime can support. If the view has `fovMutable` set to `XR_FALSE` the runtime **must** set `maxMutableFov` to be the same as `recommendedFov`.

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

The XrViewConfigurationViewFovEPIC structure is an output struct which can be added to the next chain of XrViewConfigurationView to retrieve the field-of-view for that view.

```
typedef struct XrViewConfigurationViewFovEPIC {
    XrStructureType    type;
    const void*        next;
    XrFovf             recommendedFov;
    XrFovf             maxMutableFov;
} XrViewConfigurationViewFovEPIC;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- recommendedFov is the recommended field-of-view based on the current user IPD.
- maxMutableFov is the maximum field-of-view that the runtime can display.

Valid Usage (Implicit)

- The XR_EPIC_view_configuration_fov extension **must** be enabled prior to using XrViewConfigurationViewFovEPIC
- type **must** be XR_TYPE_VIEW_CONFIGURATION_VIEW_FOV_EPIC
- next **must** be NULL or a valid pointer to the next structure in a structure chain

New Functions

Issues

Version History

- Revision 2, 2020-06-04 (Jules Blok)
  - Fixed incorrect member name.
- Revision 1, 2020-03-05 (Jules Blok)
  - Initial version.

12.35. XR_FB_android_surface_swapchain_create

Name String

XR_FB_android_surface_swapchain_create

Extension Type

Instance extension

Registered Extension Number

71

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0
- Requires XR_KHR_android_surface_swapchain

Contributors

Cass Everitt, Facebook
Gloria Kennickell, Facebook
Tomislav Novak, Facebook

Overview

This extension provides support for the specification of Android Surface specific swapchain create flags.

In order to enable the functionality of this extension, you **must** pass the name of the extension into xrCreateInstance via the XrInstanceCreateInfo `enabledExtensionNames` parameter as indicated in the Extensions section.

To specify the additional create flags, you must create a XrAndroidSurfaceSwapchainCreateInfoFB structure and pass it on the call to xrCreateSwapchainAndroidSurfaceKHR via the XrSwapchainCreateInfo structure's `next` parameter.

New Object Types

New Flag Types

```
typedef XrFlags64 XrAndroidSurfaceSwapchainFlagsFB;
```

```
// Flag bits for XrAndroidSurfaceSwapchainFlagsFB
static const XrAndroidSurfaceSwapchainFlagsFB
XR_ANDROID_SURFACE_SWAPCHAIN_SYNCHRONOUS_BIT_FB = 0x00000001;
static const XrAndroidSurfaceSwapchainFlagsFB
XR_ANDROID_SURFACE_SWAPCHAIN_USE_TIMESTAMPS_BIT_FB = 0x00000002;
```

Flag Descriptions

- XR_ANDROID_SURFACE_SWAPCHAIN_SYNCHRONOUS_BIT_FB indicates the underlying BufferQueue should be created in synchronous mode, allowing multiple buffers to be queued instead of always replacing the last buffer. Buffers are retired in order, and the producer may block until a new buffer is available.
- XR_ANDROID_SURFACE_SWAPCHAIN_USE_TIMESTAMPS_BIT_FB indicates the compositor should acquire the most recent buffer whose presentation timestamp is not greater than the expected display time of the final composited frame.

New Enum Constants

XrStructureType enumeration is extended with:

- XR_TYPE_ANDROID_SURFACE_SWAPCHAIN_CREATE_INFO_FB

New Enums

- XR_ANDROID_SURFACE_SWAPCHAIN_SYNCHRONOUS_BIT_FB
- XR_ANDROID_SURFACE_SWAPCHAIN_USE_TIMESTAMPS_BIT_FB

New Structures

The XrAndroidSurfaceSwapchainCreateInfoFB structure is defined as:

```
typedef struct XrAndroidSurfaceSwapchainCreateInfoFB {
    XrStructureType                     type;
    const void*                         next;
    XrAndroidSurfaceSwapchainFlagsFB    createFlags;
} XrAndroidSurfaceSwapchainCreateInfoFB;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- createFlags is 0 or one or more XrAndroidSurfaceSwapchainFlagBitsFB which indicate various characteristics desired for the Android Surface Swapchain.

XrAndroidSurfaceSwapchainCreateInfoFB contains additional Android Surface specific create flags when calling xrCreateSwapchainAndroidSurfaceKHR. The XrAndroidSurfaceSwapchainCreateInfoFB structure **must** be provided in the `next` chain of the XrSwapchainCreateInfo structure when calling xrCreateSwapchainAndroidSurfaceKHR.

Valid Usage (Implicit)

- The `XR_FB_android_surface_swapchain_create` extension **must** be enabled prior to using XrAndroidSurfaceSwapchainCreateInfoFB
- `type` **must** be `XR_TYPE_ANDROID_SURFACE_SWAPCHAIN_CREATE_INFO_FB`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `createFlags` **must** be a valid combination of XrAndroidSurfaceSwapchainFlagBitsFB values
- `createFlags` **must** not be `0`

New Functions

Issues

Version History

- Revision 1, 2020-12-10 (Gloria Kennickell)
  - Initial draft

12.36. XR_FB_color_space

Name String

`XR_FB_color_space`

Extension Type

Instance extension

Registered Extension Number

109

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Contributors

Volga Aksoy, Facebook
Cass Everitt, Facebook

Gloria Kennickell, Facebook

Overview

XR devices may use a color space that is different from many monitors used in development. Application developers may desire to specify the color space in which they have authored their application so appropriate colors are shown when the application is running on the XR device.

This extension allows:

- An application to get the native color space of the XR device.
- An application to enumerate the supported color spaces for the session.
- An application to set the color space for the session.

In order to enable the functionality of this extension, you **must** pass the name of the extension into xrCreateInstance via the XrInstanceCreateInfo `enabledExtensionNames` parameter as indicated in the Extensions section.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_SYSTEM_COLOR_SPACE_PROPERTIES_FB`

XrResult enumeration is extended with:

- `XR_ERROR_COLOR_SPACE_UNSUPPORTED_FB`

New Enums

The possible color spaces are specified by the XrColorSpaceFB enumeration.

```
typedef enum XrColorSpaceFB {
    XR_COLOR_SPACE_UNMANAGED_FB = 0,
    XR_COLOR_SPACE_REC2020_FB = 1,
    XR_COLOR_SPACE_REC709_FB = 2,
    XR_COLOR_SPACE_RIFT_CV1_FB = 3,
    XR_COLOR_SPACE_RIFT_S_FB = 4,
    XR_COLOR_SPACE_QUEST_FB = 5,
    XR_COLOR_SPACE_P3_FB = 6,
    XR_COLOR_SPACE_ADOBE_RGB_FB = 7,
    XR_COLOR_SPACE_MAX_ENUM_FB = 0x7FFFFFFF
} XrColorSpaceFB;
```

Enumerant Descriptions

- `XR_COLOR_SPACE_UNMANAGED_FB`. No color correction, not recommended for production use.
- `XR_COLOR_SPACE_REC2020_FB`. Standard Rec. 2020 chromacities. This is the preferred color space for standardized color across all Oculus HMDs with D65 white point.
- `XR_COLOR_SPACE_REC709_FB`. Standard Rec. 709 chromaticities, similar to sRGB.
- `XR_COLOR_SPACE_RIFT_CV1_FB`. Unique color space, between P3 and Adobe RGB using D75 white point.

  Color Space Details with Chromacity Primaries in CIE 1931 xy:

  - Red: (0.666, 0.334)
  - Green: (0.238, 0.714)
  - Blue: (0.139, 0.053)
  - White: (0.298, 0.318)
- `XR_COLOR_SPACE_RIFT_S_FB`. Unique color space. Similar to Rec 709 using D75.

  Color Space Details with Chromacity Primaries in CIE 1931 xy:

  - Red: (0.640, 0.330)
  - Green: (0.292, 0.586)
  - Blue: (0.156, 0.058)
  - White: (0.298, 0.318)
- `XR_COLOR_SPACE_QUEST_FB`. Unique color space. Similar to Rift CV1 using D75 white point

  Color Space Details with Chromacity Primaries in CIE 1931 xy:

  - Red: (0.661, 0.338)
  - Green: (0.228, 0.718)
  - Blue: (0.142, 0.042)
  - White: (0.298, 0.318)
- `XR_COLOR_SPACE_P3_FB`. Similar to DCI-P3, but uses D65 white point instead.

  Color Space Details with Chromacity Primaries in CIE 1931 xy:

  - Red: (0.680, 0.320)
  - Green: (0.265, 0.690)
  - Blue: (0.150, 0.060)
  - White: (0.313, 0.329)

- XR_COLOR_SPACE_ADOBE_RGB_FB. Standard Adobe chromacities.

New Structures

An application may inspect the native color space of the system by chaining an XrSystemColorSpacePropertiesFB structure to the XrSystemProperties when calling xrGetSystemProperties.

The XrSystemColorSpacePropertiesFB structure is defined as:

```
typedef struct XrSystemColorSpacePropertiesFB {
    XrStructureType    type;
    void*              next;
    XrColorSpaceFB     colorSpace;
} XrSystemColorSpacePropertiesFB;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- colorSpace is the native color space of the XR device.

Valid Usage (Implicit)

- The XR_FB_color_space extension **must** be enabled prior to using XrSystemColorSpacePropertiesFB
- type **must** be XR_TYPE_SYSTEM_COLOR_SPACE_PROPERTIES_FB
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- colorSpace **must** be a valid XrColorSpaceFB value

New Functions

The xrEnumerateColorSpacesFB function is defined as:

```
XrResult xrEnumerateColorSpacesFB(
    XrSession                                   session,
    uint32_t                                    colorSpaceCapacityInput,
    uint32_t*                                   colorSpaceCountOutput,
    XrColorSpaceFB*                             colorSpaces);
```

Parameter Descriptions

- `session` is the session that enumerates the supported color spaces.
- `colorSpaceCapacityInput` is the capacity of the `colorSpaces` array, or 0 to retrieve the required capacity.
- `colorSpaceCountOutput` is a pointer to the count of XrColorSpaceFB `colorSpaces` written, or a pointer to the required capacity in the case that `colorSpaceCapacityInput` is `0`.
- `colorSpaces` is a pointer to an array of XrColorSpaceFB color spaces, but **can** be `NULL` if `colorSpaceCapacityInput` is `0`.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required `colorSpaces` size.

xrEnumerateColorSpacesFB enumerates the color spaces supported by the current session. Runtimes **must** always return identical buffer contents from this enumeration for the lifetime of the session.

Valid Usage (Implicit)

- The `XR_FB_color_space` extension **must** be enabled prior to calling xrEnumerateColorSpacesFB
- `session` **must** be a valid XrSession handle
- `colorSpaceCountOutput` **must** be a pointer to a `uint32_t` value
- If `colorSpaceCapacityInput` is not `0`, `colorSpaces` **must** be a pointer to an array of `colorSpaceCapacityInput` XrColorSpaceFB values

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_SIZE_INSUFFICIENT
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE

The xrSetColorSpaceFB function is defined as:

```
XrResult xrSetColorSpaceFB(
    XrSession                                   session,
    const XrColorSpaceFB                        colorspace);
```

Parameter Descriptions

- session is a valid XrSession handle.
- colorSpace is a supported color space. Supported color spaces are indicated by xrEnumerateColorSpacesFB.

xrSetColorSpaceFB provides a mechanism for an application to specify the color space used in the final rendered frame. If this function is not called, the session will use the color space deemed appropriate by the runtime. Facebook HMDs for both PC and Mobile product lines default to XR_COLOR_SPACE_RIFT_CV1_FB. The runtime **must** return XR_ERROR_COLOR_SPACE_UNSUPPORTED_FB if colorSpace is not one of the values enumerated by xrEnumerateColorSpacesFB.

Valid Usage (Implicit)

- The XR_FB_color_space extension **must** be enabled prior to calling xrSetColorSpaceFB
- session **must** be a valid XrSession handle
- colorspace **must** be a valid XrColorSpaceFB value

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_COLOR_SPACE_UNSUPPORTED_FB
- XR_ERROR_FEATURE_UNSUPPORTED
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE

Issues

Version History

- Revision 1, 2020-11-09 (Gloria Kennickell)
  - Initial extension description

12.37. XR_FB_display_refresh_rate

Name String

XR_FB_display_refresh_rate

Extension Type

Instance extension

Registered Extension Number

102

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Contributors

Cass Everitt, Facebook
Gloria Kennickell, Facebook

Overview

On platforms which support dynamically adjusting the display refresh rate, application developers may request a specific display refresh rate in order to improve the overall user experience, examples include:

- A video application may choose a display refresh rate which better matches the video content playback rate in order to achieve smoother video frames.
- An application which can support a higher frame rate may choose to render at the higher rate to improve the overall perceptual quality, for example, lower latency and less flicker.

This extension allows:

- An application to identify what display refresh rates the session supports and the current display refresh rate.
- An application to request a display refresh rate to indicate its preference to the runtime.
- An application to receive notification of changes to the display refresh rate which are delivered via events.

In order to enable the functionality of this extension, you **must** pass the name of the extension into xrCreateInstance via the XrInstanceCreateInfo `enabledExtensionNames` parameter as indicated in the Extensions section.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_EVENT_DATA_DISPLAY_REFRESH_RATE_CHANGED_FB`

XrResult enumeration is extended with:

- XR_ERROR_DISPLAY_REFRESH_RATE_UNSUPPORTED_FB

New Enums

New Structures

Receiving the XrEventDataDisplayRefreshRateChangedFB event structure indicates that the display refresh rate has changed.

The XrEventDataDisplayRefreshRateChangedFB structure is defined as:

```
typedef struct XrEventDataDisplayRefreshRateChangedFB {
    XrStructureType    type;
    const void*        next;
    float              fromDisplayRefreshRate;
    float              toDisplayRefreshRate;
} XrEventDataDisplayRefreshRateChangedFB;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- fromDisplayRefreshRate is the previous display refresh rate.
- toDisplayRefreshRate is the new display refresh rate.

Valid Usage (Implicit)

- The XR_FB_display_refresh_rate extension **must** be enabled prior to using XrEventDataDisplayRefreshRateChangedFB
- type **must** be XR_TYPE_EVENT_DATA_DISPLAY_REFRESH_RATE_CHANGED_FB
- next **must** be NULL or a valid pointer to the next structure in a structure chain

New Functions

The xrEnumerateDisplayRefreshRatesFB function is defined as:

```
XrResult xrEnumerateDisplayRefreshRatesFB(
    XrSession                                   session,
    uint32_t                                    displayRefreshRateCapacityInput,
    uint32_t*                                   displayRefreshRateCountOutput,
    float*                                      displayRefreshRates);
```

Parameter Descriptions

- `session` is the session that enumerates the supported display refresh rates.
- `displayRefreshRateCapacityInput` is the capacity of the `displayRefreshRates`, or 0 to retrieve the required capacity.
- `displayRefreshRateCountOutput` is a pointer to the count of `float` `displayRefreshRates` written, or a pointer to the required capacity in the case that `displayRefreshRateCapacityInput` is `0`.
- `displayRefreshRates` is a pointer to an array of `float` display refresh rates, but **can** be `NULL` if `displayRefreshRateCapacityInput` is `0`.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required `displayRefreshRates` size.

xrEnumerateDisplayRefreshRatesFB enumerates the display refresh rates supported by the current session. Display refresh rates **must** be in order from lowest to highest supported display refresh rates. Runtimes **must** always return identical buffer contents from this enumeration for the lifetime of the session.

Valid Usage (Implicit)

- The `XR_FB_display_refresh_rate` extension **must** be enabled prior to calling xrEnumerateDisplayRefreshRatesFB
- `session` **must** be a valid XrSession handle
- `displayRefreshRateCountOutput` **must** be a pointer to a `uint32_t` value
- If `displayRefreshRateCapacityInput` is not `0`, `displayRefreshRates` **must** be a pointer to an array of `displayRefreshRateCapacityInput` `float` values

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_SIZE_INSUFFICIENT
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE

The xrGetDisplayRefreshRateFB function is defined as:

```
XrResult xrGetDisplayRefreshRateFB(
    XrSession                                   session,
    float*                                      displayRefreshRate);
```

Parameter Descriptions

- session is the XrSession to query.
- displayRefreshRate is a pointer to a float into which the current display refresh rate will be placed.

xrGetDisplayRefreshRateFB retrieves the current display refresh rate.

Valid Usage (Implicit)

- The XR_FB_display_refresh_rate extension **must** be enabled prior to calling xrGetDisplayRefreshRateFB
- session **must** be a valid XrSession handle
- displayRefreshRate **must** be a pointer to a float value

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE

The xrRequestDisplayRefreshRateFB function is defined as:

```
XrResult xrRequestDisplayRefreshRateFB(
    XrSession                                   session,
    float                                       displayRefreshRate);
```

Parameter Descriptions

- session is a valid XrSession handle.
- displayRefreshRate is 0.0f or a supported display refresh rate. Supported display refresh rates are indicated by xrEnumerateDisplayRefreshRatesFB.

xrRequestDisplayRefreshRateFB provides a mechanism for an application to request the system to dynamically change the display refresh rate to the application preferred value. The runtime **must** return XR_ERROR_DISPLAY_REFRESH_RATE_UNSUPPORTED_FB if displayRefreshRate is not either 0.0f or one of the values enumerated by xrEnumerateDisplayRefreshRatesFB. A display refresh rate of 0.0f indicates the application has no preference.

Note that this is only a request and does not guarantee the system will switch to the requested display refresh rate.

Valid Usage (Implicit)

- The `XR_FB_display_refresh_rate` extension **must** be enabled prior to calling xrRequestDisplayRefreshRateFB
- `session` **must** be a valid XrSession handle

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_DISPLAY_REFRESH_RATE_UNSUPPORTED_FB`
- `XR_ERROR_FEATURE_UNSUPPORTED`
- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`

Issues

Changing the display refresh rate from its system default does not come without trade-offs. Increasing the display refresh rate puts more load on the entire system and can lead to thermal degradation. Conversely, lowering the display refresh rate can provide better thermal sustainability but at the cost of more perceptual issues, like higher latency and flickering.

Version History

- Revision 1, 2020-10-05 (Gloria Kennickell)
  - Initial extension description

12.38. XR_FB_swapchain_update_state

Name String

`XR_FB_swapchain_update_state`

Extension Type

Instance extension

Registered Extension Number

72

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Contributors

Cass Everitt, Facebook
Gloria Kennickell, Facebook

Overview

This extension enables the application to modify specific mutable state associated with a swapchain, examples include:

- A video application may need to update the default size of the image buffers associated with an Android Surface Swapchain.
- A video application may need to communicate a new width and height for an Android Surface Swapchain, as the surface dimensions may be implicitly updated by the producer during the life of the Swapchain. This is important for correct application of the non-normalized `imageRect` specified via XrSwapchainSubImage.
- On platforms where composition runs in a separate process from the application, swapchains must be created in a cross-process friendly way. In such cases, the texture image memory may be shared between processes, but the texture state may not; and, an explicit mechanism to synchronize this texture state between the application and the compositor is required.

In order to enable the functionality of this extension, you **must** pass the name of the extension into xrCreateInstance via the XrInstanceCreateInfo `enabledExtensionNames` parameter as indicated in the Extensions section.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_SWAPCHAIN_STATE_ANDROID_SURFACE_DIMENSIONS_FB`

- XR_TYPE_SWAPCHAIN_STATE_SAMPLER_OPENGL_ES_FB

New Enums

New Structures

The XrSwapchainStateBaseHeaderFB structure is defined as:

```
typedef struct XrSwapchainStateBaseHeaderFB {
    XrStructureType    type;
    void*              next;
} XrSwapchainStateBaseHeaderFB;
```

Member Descriptions

- type is the XrStructureType of this structure. This base structure itself has no associated XrStructureType value.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.

The XrSwapchainStateBaseHeaderFB is a base structure that can be overridden by a specific XrSwapchainState* child structure.

Valid Usage (Implicit)

- The XR_FB_swapchain_update_state extension **must** be enabled prior to using XrSwapchainStateBaseHeaderFB
- type **must** be one of the following XrStructureType values: XR_TYPE_SWAPCHAIN_STATE_ANDROID_SURFACE_DIMENSIONS_FB, XR_TYPE_SWAPCHAIN_STATE_SAMPLER_OPENGL_ES_FB
- next **must** be NULL or a valid pointer to the next structure in a structure chain

These structures are only available when the corresponding XR_USE_PLATFORM_ macro is defined before including openxr_platform.h.

The XrSwapchainStateAndroidSurfaceDimensionsFB structure is defined as:

```
typedef struct XrSwapchainStateAndroidSurfaceDimensionsFB {
    XrStructureType    type;
    const void*        next;
    uint32_t           width;
    uint32_t           height;
} XrSwapchainStateAndroidSurfaceDimensionsFB;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- width is the width of the image buffer, must not be greater than the graphics API's maximum limit.
- height is the height of the image buffer, must not be greater than the graphics API's maximum limit.

When XrSwapchainStateAndroidSurfaceDimensionsFB is specified in the call to xrUpdateSwapchainFB, the dimensions provided will be used to update the default size of the image buffers associated with the Android Surface swapchain.

Additionally, the dimensions provided will become the new source of truth for the swapchain width and height, affecting operations such as computing the normalized imageRect for the swapchain.

To use XrSwapchainStateAndroidSurfaceDimensionsFB, XR_USE_PLATFORM_ANDROID must be defined before including openxr_platform.h.

Valid Usage (Implicit)

- The XR_FB_swapchain_update_state extension **must** be enabled prior to using XrSwapchainStateAndroidSurfaceDimensionsFB
- type **must** be XR_TYPE_SWAPCHAIN_STATE_ANDROID_SURFACE_DIMENSIONS_FB
- next **must** be NULL or a valid pointer to the next structure in a structure chain

The XrSwapchainStateSamplerOpenGLESFB structure is defined as:

```
typedef struct XrSwapchainStateSamplerOpenGLESFB {
    XrStructureType    type;
    const void*        next;
    EGLenum            minFilter;
    EGLenum            magFilter;
    EGLenum            wrapModeS;
    EGLenum            wrapModeT;
    EGLenum            swizzleRed;
    EGLenum            swizzleGreen;
    EGLenum            swizzleBlue;
    EGLenum            swizzleAlpha;
    float              maxAnisotropy;
    XrColor4f          borderColor;
} XrSwapchainStateSamplerOpenGLESFB;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `minFilter` is a valid Android OpenGL ES `EGLenum`.
- `magFilter` is a valid Android OpenGL ES `EGLenum`.
- `wrapModeS` is a valid Android OpenGL ES `EGLenum`.
- `wrapModeT` is a valid Android OpenGL ES `EGLenum`.
- `swizzleRed` is a valid Android OpenGL ES `EGLenum`.
- `swizzleGreen` is a valid Android OpenGL ES `EGLenum`.
- `swizzleBlue` is a valid Android OpenGL ES `EGLenum`.
- `swizzleAlpha` is a valid Android OpenGL ES `EGLenum`.
- `maxAnisotropy` is a valid float used to represent max anisotropy.
- `borderColor` is an RGBA color to be used as border texels.

When XrSwapchainStateSamplerOpenGLESFB is specified in the call to xrUpdateSwapchainFB, all images in the XrSwapchain will have the specified sampler state applied and the state synchronized with the compositor. Applications are expected to handle synchronization of the sampler state update with application-side rendering. Similarly, the compositor will synchronize the sampler state update with rendering of the next compositor frame. An `EGLContext`, either the `EGLContext` bound during XrSwapchain creation or an `EGLContext` in the same share group, is required to be bound on the application calling thread. Current texture bindings may be altered by the call, including the active texture.

To use XrSwapchainStateSamplerOpenGLESFB, XR_USE_PLATFORM_ANDROID and XR_USE_GRAPHICS_API_OPENGL_ES must both be defined before including openxr_platform.h.

Valid Usage (Implicit)

- The XR_FB_swapchain_update_state extension **must** be enabled prior to using XrSwapchainStateSamplerOpenGLESFB
- type **must** be XR_TYPE_SWAPCHAIN_STATE_SAMPLER_OPENGL_ES_FB
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- minFilter **must** be a valid EGLenum value
- magFilter **must** be a valid EGLenum value
- wrapModeS **must** be a valid EGLenum value
- wrapModeT **must** be a valid EGLenum value
- swizzleRed **must** be a valid EGLenum value
- swizzleGreen **must** be a valid EGLenum value
- swizzleBlue **must** be a valid EGLenum value
- swizzleAlpha **must** be a valid EGLenum value

New Functions

The xrUpdateSwapchainFB function is defined as:

```
XrResult xrUpdateSwapchainFB(
    XrSwapchain                                 swapchain,
    const XrSwapchainStateBaseHeaderFB*         state);
```

Parameter Descriptions

- swapchain is the XrSwapchain to update state for.
- state is a pointer to a XrSwapchainState structure based off of XrSwapchainStateBaseHeaderFB.

xrUpdateSwapchainFB provides support for an application to update specific mutable state associated with an XrSwapchain.

To use xrUpdateSwapchainFB, XR_USE_PLATFORM_ANDROID must be defined before including openxr_platform.h.

Valid Usage (Implicit)

- The XR_FB_swapchain_update_state extension **must** be enabled prior to calling xrUpdateSwapchainFB
- swapchain **must** be a valid XrSwapchain handle
- state **must** be a pointer to a valid XrSwapchainStateBaseHeaderFB-based structure. See also: XrSwapchainStateAndroidSurfaceDimensionsFB, XrSwapchainStateSamplerOpenGLESFB

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE

Issues

- Given we allow mutable state to be updated by the application, should we provide a mechanism to query the current state for all update structures?

Version History

- Revision 1, 2021-04-16 (Gloria Kennickell)
  - Initial extension description

12.39. XR_HTC_vive_cosmos_controller_interaction

Name String

XR_HTC_vive_cosmos_controller_interaction

Extension Type

Instance extension

Registered Extension Number

103

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2020-09-28

IP Status

No known IP claims.

Contributors

Chris Kuo, HTC
Kyle Chen, HTC

Overview

This extension defines a new interaction profile for the VIVE Cosmos Controller.

VIVE Cosmos Controller interaction profile

Interaction profile path:

- */interaction_profiles/htc/vive_cosmos_controller*

Valid for user paths:

- */user/hand/left*
- */user/hand/right*

This interaction profile represents the input sources and haptics on the VIVE Cosmos Controller.

Supported component paths:

- On */user/hand/left* only:
    - *…/input/x/click*
    - *…/input/y/click*
    - *…/input/menu/click*

- On */user/hand/right* only:
  - *…/input/a/click*
  - *…/input/b/click*
  - *…/input/system/click* (**may** not be available for application use)
- *…/input/shoulder/click*
- *…/input/squeeze/click*
- *…/input/trigger/click*
- *…/input/trigger/value*
- *…/input/thumbstick/x*
- *…/input/thumbstick/y*
- *…/input/thumbstick/click*
- *…/input/thumbstick/touch*
- *…/input/grip/pose*
- *…/input/aim/pose*
- *…/output/haptic*

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

New Functions

Issues

Version History

- Revision 1, 2020-09-28 (Chris Kuo)
  - Initial extension description

12.40. XR_HUAWEI_controller_interaction

Name String

`XR_HUAWEI_controller_interaction`

Extension Type

Instance extension

Registered Extension Number

70

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2020-05-26

IP Status

No known IP claims.

Contributors

Guodong Chen, Huawei
Kai Shao, Huawei
Yang Tao, Huawei
Gang Shen, Huawei
Yihong Huang, Huawei

Overview

This extension defines a new interaction profile for the Huawei Controller, including but not limited to Huawei VR Glasses Controller.

Huawei Controller interaction profile

Interaction profile path:

- */interaction_profiles/huawei/controller*

Valid for user paths:

- */user/hand/left*
- */user/hand/right*

This interaction profile represents the input sources and haptics on the Huawei Controller.

Supported component paths:

- *…/input/home/click*

- *…/input/back/click*
- *…/input/volume_up/click*
- *…/input/volume_down/click*
- *…/input/trigger/value*
- *…/input/trigger/click*
- *…/input/trackpad/x*
- *…/input/trackpad/y*
- *…/input/trackpad/click*
- *…/input/trackpad/touch*
- *…/input/aim/pose*
- *…/input/grip/pose*
- *…/output/haptic*

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

New Functions

Issues

Version History

- Revision 1, 2020-04-28 (Yihong Huang)
    - Initial extension description

12.41. XR_MND_headless

Name String

`XR_MND_headless`

Extension Type

Instance extension

Registered Extension Number

43

Revision

2

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2019-10-22

IP Status

No known IP claims.

Contributors

Ryan Pavlik, Collabora

Overview

Some applications may wish to access XR interaction devices without presenting any image content on the display(s). This extension provides a mechanism for writing such an application using the OpenXR API. It modifies the specification in the following ways, without adding any new named entities.

- When this extension is enabled, an application **may** call xrCreateSession without an `XrGraphicsBinding*` structure in its `next` chain. In this case, the runtime **must** create a "headless" session that does not interact with the display.
- In a headless session, the session state **should** proceed to `XR_SESSION_STATE_READY` directly from `XR_SESSION_STATE_IDLE`.
- In a headless session, the XrSessionBeginInfo::`primaryViewConfigurationType` **must** be ignored and **may** be `0`.
- In a headless session, the session state proceeds to `XR_SESSION_STATE_SYNCHRONIZED`, then `XR_SESSION_STATE_VISIBLE` and `XR_SESSION_STATE_FOCUSED`, after the call to xrBeginSession. The application does not need to call xrWaitFrame, xrBeginFrame, or xrEndFrame, unlike with non-headless sessions.
- In a headless session, xrEnumerateSwapchainFormats **must** return `XR_SUCCESS` but enumerate `0` formats.
- xrWaitFrame **must** set XrFrameState::`shouldRender` to `XR_FALSE` in a headless session. The VISIBLE and FOCUSED states are only used for their input-related semantics, not their rendering-related semantics, and these functions are permitted to allow minimal change between headless and non-headless code if desired.

Because xrWaitFrame is not required, an application using a headless session **should** sleep periodically to avoid consuming all available system resources in a busy-wait loop.

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

New Functions

Issues

- Not all devices with which this would be useful fit into one of the existing XrFormFactor values.

Version History

- Revision 1, 2019-07-25 (Ryan Pavlik)
  - Initial version reflecting Monado prototype.
- Revision 2, 2019-10-22 (Ryan Pavlik)
  - Clarify that `xrWaitFrame` is permitted and should set `shouldRender` to false.

12.42. XR_MND_swapchain_usage_input_attachment_bit

Name String

`XR_MND_swapchain_usage_input_attachment_bit`

Extension Type

Instance extension

Registered Extension Number

97

Revision

2

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2020-07-24

IP Status

No known IP claims.

Contributors

Jakob Bornecrantz, Collabora

Overview

This extension enables application to specify that swapchain images should be created in a way so that they can be used as input attachments. At the time of writing this bit only affects Vulkan swapchains.

New Object Types

New Flag Types

New Enum Constants

XrSwapchainUsageFlagBits enumeration is extended with:

- `XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_MND`

New Enums

New Structures

New Functions

Issues

Version History

- Revision 1, 2020-07-23 (Jakob Bornecrantz)
  - Initial draft
- Revision 2, 2020-07-24 (Jakob Bornecrantz)
  - Added note about only affecting Vulkan
  - Changed from MNDX to MND

12.43. XR_MSFT_controller_model

Name String

`XR_MSFT_controller_model`

Extension Type

Instance extension

Registered Extension Number

56

Revision

2

Extension and Version Dependencies

- Requires OpenXR 1.0

Contributors

Bryce Hutchings, Microsoft
Darryl Gough, Microsoft
Yin Li, Microsoft
Lachlan Ford, Microsoft

Overview

This extension provides a mechanism to load a GLTF model for controllers. An application **can** render the controller model using the real time pose input from controller's grip action pose and animate controller parts representing the user's interactions, such as pressing a button, or pulling a trigger.

This extension supports any controller interaction profile that supports *…/grip/pose*. The returned controller model represents the physical controller held in the user's hands, and it **may** be different from the current interaction profile.

Query controller model key

xrGetControllerModelKeyMSFT retrieves the `XrControllerModelKeyMSFT` for a controller. This model key **may** later be used to retrieve the model data.

The xrGetControllerModelKeyMSFT function is defined as:

```
XrResult xrGetControllerModelKeyMSFT(
    XrSession                                   session,
    XrPath                                      topLevelUserPath,
    XrControllerModelKeyStateMSFT*              controllerModelKeyState);
```

Parameter Descriptions

- `session` is the specified XrSession.
- `topLevelUserPath` is the top level user path corresponding to the controller render model being queried (e.g. */user/hand/left* or */user/hand/right*).
- `controllerModelKeyState` is a pointer to the XrControllerModelKeyStateMSFT to write the model key state to.

Valid Usage (Implicit)

- The `XR_MSFT_controller_model` extension **must** be enabled prior to calling xrGetControllerModelKeyMSFT
- `session` **must** be a valid XrSession handle
- `controllerModelKeyState` **must** be a pointer to an XrControllerModelKeyStateMSFT structure

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_OUT_OF_MEMORY`
- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_CONTROLLER_MODEL_KEY_INVALID_MSFT`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_SESSION_LOST`
- `XR_SESSION_LOSS_PENDING`
- `XR_ERROR_PATH_UNSUPPORTED`
- `XR_ERROR_PATH_INVALID`

The XrControllerModelKeyStateMSFT structure is defined as:

```
typedef struct XrControllerModelKeyStateMSFT {
    XrStructureType            type;
    void*                      next;
    XrControllerModelKeyMSFT   modelKey;
} XrControllerModelKeyStateMSFT;
```

Parameter Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain.
- modelKey is the model key corresponding to the controller render model being queried.

The modelKey value for the session represents a unique controller model that can be retrieved from xrLoadControllerModelMSFT function. Therefore, the application **can** use modelKey to cache the returned data from xrLoadControllerModelMSFT for the session.

A modelKey value of XR_NULL_CONTROLLER_MODEL_KEY_MSFT, represents an invalid model key and indicates there is no controller model yet available. The application **should** keep calling xrGetControllerModelKeyMSFT because the model **may** become available at a later point.

Valid Usage (Implicit)

- The XR_MSFT_controller_model extension **must** be enabled prior to using XrControllerModelKeyStateMSFT
- type **must** be XR_TYPE_CONTROLLER_MODEL_KEY_STATE_MSFT
- next **must** be NULL or a valid pointer to the next structure in a structure chain

```
#define XR_NULL_CONTROLLER_MODEL_KEY_MSFT 0
```

XR_NULL_CONTROLLER_MODEL_KEY_MSFT defines an invalid model key value.

```
XR_DEFINE_ATOM(XrControllerModelKeyMSFT)
```

The controller model key used to retrieve the data for the renderable controller model and associated properties and state.

Load controller model as glTF 2.0 data

Once the application obtained a valid modelKey, it **can** use the xrLoadControllerModelMSFT function to load the GLB data for the controller model.

The xrLoadControllerModelMSFT function loads the controller model as a byte buffer containing a binary form of glTF (a.k.a GLB file format) for the controller. The binary glTF data **must** conform to glTF 2.0 format defined at https://github.com/KhronosGroup/glTF/tree/master/specification/2.0.

```
XrResult xrLoadControllerModelMSFT(
    XrSession                                   session,
    XrControllerModelKeyMSFT                    modelKey,
    uint32_t                                    bufferCapacityInput,
    uint32_t*                                   bufferCountOutput,
    uint8_t*                                    buffer);
```

Parameter Descriptions

- `session` is the specified XrSession.
- `modelKey` is the model key corresponding to the controller render model being queried.
- `bufferCapacityInput` is the capacity of the `buffer` array, or 0 to indicate a request to retrieve the required capacity.
- `bufferCountOutput` filled in by the runtime with the count of elements in `buffer` array, or returns the required capacity in the case that `bufferCapacityInput` is 0.
- `buffer` is a pointer to an application-allocated array of the model for the device that will be filled with the `uint8_t` values by the runtime. It **can** be `NULL` if `bufferCapacityInput` is 0.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required `buffer` size.

The xrLoadControllerModelMSFT function **may** be a slow operation and therefore **should** be invoked from a non-timing critical thread.

If the input `modelKey` is invalid, i.e. it is XR_NULL_CONTROLLER_MODEL_KEY_MSFT or not a key returned from XrControllerModelKeyStateMSFT, the runtime **must** return `XR_ERROR_CONTROLLER_MODEL_KEY_INVALID_MSFT`.

Valid Usage (Implicit)

- The `XR_MSFT_controller_model` extension **must** be enabled prior to calling xrLoadControllerModelMSFT
- `session` **must** be a valid XrSession handle
- `bufferCountOutput` **must** be a pointer to a `uint32_t` value
- If `bufferCapacityInput` is not `0`, `buffer` **must** be a pointer to an array of `bufferCapacityInput` `uint8_t` values

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_OUT_OF_MEMORY`
- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_SIZE_INSUFFICIENT`
- `XR_ERROR_CONTROLLER_MODEL_KEY_INVALID_MSFT`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_SESSION_LOST`
- `XR_SESSION_LOSS_PENDING`

Animate controller parts

The application **can** animate parts of the glTF model to represent the user's interaction on the controller, such as pressing a button or pulling a trigger.

Once the application loads the glTF model of the controller, it **should** first get XrControllerModelPropertiesMSFT containing an array of node names in the glTF model that **can** be animated. These properties, including the order of these node names in the array, **must** be immutable for a valid `modelKey` in the session, and therefore **can** be cached. In the frame loop, the application **should** get XrControllerModelStateMSFT to retrieve the pose of each node representing user's interaction on the controller and apply the transform to the corresponding node in the glTF model using application's glTF renderer.

The xrGetControllerModelPropertiesMSFT function returns the controller model properties for a given modelKey.

```
XrResult xrGetControllerModelPropertiesMSFT(
    XrSession                                   session,
    XrControllerModelKeyMSFT                    modelKey,
    XrControllerModelPropertiesMSFT*            properties);
```

Parameter Descriptions

- session is the specified XrSession.
- modelKey is a valid model key obtained from XrControllerModelKeyStateMSFT
- properties is a XrControllerModelPropertiesMSFT returns the properties of the controller model

The runtime **must** return the same data in XrControllerModelPropertiesMSFT for a valid modelKey. Therefore, the application **can** cache the returned XrControllerModelPropertiesMSFT using modelKey and reuse the data for each frame.

If the input modelKey is invalid, i.e. it is XR_NULL_CONTROLLER_MODEL_KEY_MSFT or not a key returned from XrControllerModelKeyStateMSFT, the runtime **must** return XR_ERROR_CONTROLLER_MODEL_KEY_INVALID_MSFT.

Valid Usage (Implicit)

- The XR_MSFT_controller_model extension **must** be enabled prior to calling xrGetControllerModelPropertiesMSFT
- session **must** be a valid XrSession handle
- properties **must** be a pointer to an XrControllerModelPropertiesMSFT structure

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_CONTROLLER_MODEL_KEY_INVALID_MSFT
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_SESSION_LOST
- XR_SESSION_LOSS_PENDING

The XrControllerModelPropertiesMSFT structure describes the properties of a controller model including an array of XrControllerModelNodePropertiesMSFT.

```
typedef struct XrControllerModelPropertiesMSFT {
    XrStructureType                         type;
    void*                                   next;
    uint32_t                                nodeCapacityInput;
    uint32_t                                nodeCountOutput;
    XrControllerModelNodePropertiesMSFT*    nodeProperties;
} XrControllerModelPropertiesMSFT;
```

Parameter Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain.
- `nodeCapacityInput` is the capacity of the `nodeProperties` array, or 0 to indicate a request to retrieve the required capacity.
- `nodeCountOutput` filled in by the runtime with the count of elements in `nodeProperties` array, or returns the required capacity in the case that `nodeCapacityInput` is 0.
- `nodeProperties` is a pointer to an application-allocated array that will be filled with the XrControllerModelNodePropertiesMSFT values. It **can** be `NULL` if `nodeCapacityInput` is 0.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required `nodeProperties` size.

Valid Usage (Implicit)

- The `XR_MSFT_controller_model` extension **must** be enabled prior to using XrControllerModelPropertiesMSFT
- `type` **must** be `XR_TYPE_CONTROLLER_MODEL_PROPERTIES_MSFT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- If `nodeCapacityInput` is not `0`, `nodeProperties` **must** be a pointer to an array of `nodeCapacityInput` XrControllerModelNodePropertiesMSFT structures

The XrControllerModelNodePropertiesMSFT structure describes properties of animatable nodes, including the node name and parent node name to locate a glTF node in the controller model that **can** be animated based on user's interactions on the controller.

```
typedef struct XrControllerModelNodePropertiesMSFT {
    XrStructureType    type;
    void*              next;
    char               parentNodeName[XR_MAX_CONTROLLER_MODEL_NODE_NAME_SIZE_MSFT];
    char               nodeName[XR_MAX_CONTROLLER_MODEL_NODE_NAME_SIZE_MSFT];
} XrControllerModelNodePropertiesMSFT;
```

Parameter Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain.
- `parentNodeName` is the name of the parent node in the provided glTF file. The parent name **may** be empty if it should not be used to locate this node.
- `nodeName` is the name of this node in the provided glTF file.

The node can be located in the glTF node hierarchy by finding the node(s) with the matching node name and parent node name. If the `parentNodeName` is empty, the matching will be solely based on the `nodeName`.

If there are multiple nodes in the glTF file matches the condition above, the first matching node using depth-first traversal in the glTF scene **should** be animated and the rest **should** be ignored.

The runtime **must** not return any `nodeName` or `parentName` that doesn't match any gltTF nodes in the corresponding controller model.

Valid Usage (Implicit)

- The `XR_MSFT_controller_model` extension **must** be enabled prior to using XrControllerModelNodePropertiesMSFT
- `type` **must** be `XR_TYPE_CONTROLLER_MODEL_NODE_PROPERTIES_MSFT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `parentNodeName` **must** be a null-terminated UTF-8 string whose length is less than or equal to XR_MAX_CONTROLLER_MODEL_NODE_NAME_SIZE_MSFT
- `nodeName` **must** be a null-terminated UTF-8 string whose length is less than or equal to XR_MAX_CONTROLLER_MODEL_NODE_NAME_SIZE_MSFT

The xrGetControllerModelStateMSFT function returns the current state of the controller model representing user's interaction to the controller, such as pressing a button or pulling a trigger.

```
XrResult xrGetControllerModelStateMSFT(
    XrSession                                   session,
    XrControllerModelKeyMSFT                    modelKey,
    XrControllerModelStateMSFT*                 state);
```

Parameter Descriptions

- `session` is the specified XrSession.
- `modelKey` is the model key corresponding to the controller model being queried.
- `state` is a pointer to XrControllerModelNodeStateMSFT returns the current controller model state.

The runtime **may** return different state for a model key after each call to xrSyncActions, which represents the latest state of the user interactions.

If the input `modelKey` is invalid, i.e. it is XR_NULL_CONTROLLER_MODEL_KEY_MSFT or not a key returned from XrControllerModelKeyStateMSFT, the runtime **must** return `XR_ERROR_CONTROLLER_MODEL_KEY_INVALID_MSFT`.

Valid Usage (Implicit)

- The `XR_MSFT_controller_model` extension **must** be enabled prior to calling xrGetControllerModelStateMSFT
- `session` **must** be a valid XrSession handle
- `state` **must** be a pointer to an XrControllerModelStateMSFT structure

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_OUT_OF_MEMORY`
- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_CONTROLLER_MODEL_KEY_INVALID_MSFT`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_SESSION_LOST`
- `XR_SESSION_LOSS_PENDING`

The XrControllerModelStateMSFT structure describes the state of a controller model, including an array of XrControllerModelNodeStateMSFT.

```
typedef struct XrControllerModelStateMSFT {
    XrStructureType                    type;
    void*                              next;
    uint32_t                           nodeCapacityInput;
    uint32_t                           nodeCountOutput;
    XrControllerModelNodeStateMSFT*    nodeStates;
} XrControllerModelStateMSFT;
```

Parameter Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain.
- `nodeCapacityInput` is the capacity of the `nodeStates` array, or 0 to indicate a request to retrieve the required capacity.
- `nodeCountOutput` filled in by the runtime with the count of elements in `nodeStates` array, or returns the required capacity in the case that `nodeCapacityInput` is 0.
- `nodeStates` is a pointer to an application-allocated array that will be filled with the XrControllerModelNodeStateMSFT values. It **can** be `NULL` if `sourceCapacityInput` is 0.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required `nodeStates` size.

Valid Usage (Implicit)

- The `XR_MSFT_controller_model` extension **must** be enabled prior to using XrControllerModelStateMSFT
- `type` **must** be `XR_TYPE_CONTROLLER_MODEL_STATE_MSFT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- If `nodeCapacityInput` is not `0`, `nodeStates` **must** be a pointer to an array of `nodeCapacityInput` XrControllerModelNodeStateMSFT structures

The XrControllerModelNodeStateMSFT structure describes the state of a node in a controller model.

```
typedef struct XrControllerModelNodeStateMSFT {
    XrStructureType    type;
    void*              next;
    XrPosef            nodePose;
} XrControllerModelNodeStateMSFT;
```

Parameter Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain.
- nodePose is an XrPosef of the node in its parent node space.

The state is corresponding to the glTF node identified by the nodeName and nodeParentName of the node property at the same array index in the nodeProperties in XrControllerModelPropertiesMSFT.

The nodePose is based on the user's interaction on the controller at the latest xrSyncActions, represented as the XrPosef of the node in it's parent node space.

Valid Usage (Implicit)

- The XR_MSFT_controller_model extension **must** be enabled prior to using XrControllerModelNodeStateMSFT
- type **must** be XR_TYPE_CONTROLLER_MODEL_NODE_STATE_MSFT
- next **must** be NULL or a valid pointer to the next structure in a structure chain

New Object Types

New Flag Types

New Enum Constants

- XR_MAX_CONTROLLER_MODEL_NODE_NAME_SIZE_MSFT
- XR_TYPE_CONTROLLER_MODEL_NODE_PROPERTIES_MSFT
- XR_TYPE_CONTROLLER_MODEL_PROPERTIES_MSFT
- XR_TYPE_CONTROLLER_MODEL_NODE_STATE_MSFT
- XR_TYPE_CONTROLLER_MODEL_STATE_MSFT
- XR_ERROR_CONTROLLER_MODEL_KEY_INVALID_MSFT

New Enums

New Structures

- XrControllerModelKeyStateMSFT
- XrControllerModelNodePropertiesMSFT
- XrControllerModelPropertiesMSFT
- XrControllerModelNodeStateMSFT
- XrControllerModelStateMSFT

New Functions

- xrGetControllerModelKeyMSFT
- xrLoadControllerModelMSFT
- xrGetControllerModelPropertiesMSFT
- xrGetControllerModelStateMSFT

Issues

Version History

- Revision 1, 2020-03-12 (Yin Li)
  - Initial extension description
- Revision 2, 2020-08-12 (Bryce Hutchings)
  - Remove a possible error condition

12.44. XR_MSFT_first_person_observer

Name String

`XR_MSFT_first_person_observer`

Extension Type

Instance extension

Registered Extension Number

55

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0
- Requires `XR_MSFT_secondary_view_configuration`

Last Modified Date

2020-05-02

IP Status

No known IP claims.

Contributors

Yin Li, Microsoft
Zonglin Wu, Microsoft
Alex Turner, Microsoft

12.44.1. Overview

This first-person observer view configuration enables the runtime to request the application to render an additional first-person view of the scene to be composed onto video frames being captured from a camera attached to and moved with the primary display on the form factor, which is generally for viewing on a 2D screen by an external observer. This first-person camera will be facing forward with roughly the same perspective as the primary views, and so the application should render its view to show objects that surround the user and avoid rendering the user's body avatar. The runtime is responsible for composing the application's rendered observer view onto the camera frame based on the chosen environment blend mode for this view configuration, as this extension does not provide the associated camera frame to the application.

This extension requires the XR_MSFT_secondary_view_configuration extension to also be enabled.

`XR_VIEW_CONFIGURATION_TYPE_SECONDARY_MONO_FIRST_PERSON_OBSERVER_MSFT` requires one element in XrViewConfigurationProperties and one projection in each XrCompositionLayerProjection layer.

Runtimes **should** only make this view configuration active when the user or the application activates a runtime feature that will make use of the resulting composed camera frames, for example taking a mixed reality photo. Otherwise, the runtime **should** leave this view configuration inactive to avoid the application wasting CPU and GPU resources rendering unnecessarily for this extra view.

Because this is a first-person view of the scene, applications **can** share a common culling and instanced rendering pass with their primary view renders. However, the view state (pose and FOV) of the first-person observer view will not match the view state of any of the primary views. Applications enabling this view configuration **must** call xrLocateViews a second time each frame to explicitly query the view state for the `XR_VIEW_CONFIGURATION_TYPE_SECONDARY_MONO_FIRST_PERSON_OBSERVER_MSFT` configuration.

This secondary view configuration **may** support a different set of environment blend modes than the primary view configuration. For example, a device that only supports additive blending for its primary display may support alpha-blending when composing the first-person observer view with camera frames. The application should render with assets and shaders that produce output acceptable to both the primary and observer view configuration's environment blend modes when sharing render passes across both view configurations.

New Object Types

New Flag Types

New Enum Constants

XrViewConfigurationType enumeration is extended with:

- `XR_VIEW_CONFIGURATION_TYPE_SECONDARY_MONO_FIRST_PERSON_OBSERVER_MSFT`

New Enums

New Structures

New Functions

Issues

Version History

- Revision 1, 2019-07-30 (Yin LI)
  - Initial extension description

12.45. XR_MSFT_hand_interaction

Name String

`XR_MSFT_hand_interaction`

Extension Type

Instance extension

Registered Extension Number

51

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Contributors

Yin Li, Microsoft
Lachlan Ford, Microsoft
Alex Turner, Microsoft

Overview

This extension defines a new interaction profile for near interactions and far interactions driven by directly-tracked hands.

Hand interaction profile

Interaction profile path:

- */interaction_profiles/microsoft/hand_interaction*

Valid for top level user path:

- */user/hand/left*
- */user/hand/right*

This interaction profile provides basic pose and actions for near and far interactions using hand tracking input.

Supported component paths:

- *…/input/select/value*
- *…/input/squeeze/value*
- *…/input/aim/pose*
- *…/input/grip/pose*

The application **should** use the *…/select/value* and *…/aim/pose* paths for far hand interactions, such as using a virtual laser pointer to target and click a button on the wall. Here, *…/select/value* **can** be used as either a boolean or float action type, where the value `XR_TRUE` or `1.0f` represents a closed hand shape.

The application **should** use the *…/squeeze/value* and *…/grip/pose* for near hand interactions, such as picking up a virtual object within the user's reach from a table. Here, *…/squeeze/value* **can** be used as either a boolean or float action type, where the value `XR_TRUE` or `1.0f` represents a closed hand shape.

The runtime **may** trigger both "select" and "squeeze" actions for the same hand gesture if the user's hand gesture is able to trigger both near and far interactions. The application **should** not assume they are as independent as two buttons on a controller.

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

New Functions

Issues

Version History

- Revision 1, 2019-09-16 (Yin Li)
  - Initial extension description

12.46. XR_MSFT_hand_tracking_mesh

Name String

`XR_MSFT_hand_tracking_mesh`

Extension Type

Instance extension

Registered Extension Number

53

Revision

3

Extension and Version Dependencies

- Requires OpenXR 1.0
- Requires `XR_EXT_hand_tracking`

Last Modified Date

2021-04-13

IP Status

No known IP claims.

Contributors

Yin Li, Microsoft
Lachlan Ford, Microsoft
Alex Turner, Microsoft
Bryce Hutchings, Microsoft

Overview

This extension enables hand tracking inputs represented as a dynamic hand mesh. It enables applications to render hands in XR experiences and interact with virtual objects using hand meshes.

The application **must** also enable the `XR_EXT_hand_tracking` extension in order to use this extension.

Inspect system capability

An application **can** inspect whether the system is capable of hand tracking meshes by chaining a XrSystemHandTrackingMeshPropertiesMSFT structure to the XrSystemProperties when calling xrGetSystemProperties.

```
typedef struct XrSystemHandTrackingMeshPropertiesMSFT {
    XrStructureType    type;
    void*              next;
    XrBool32           supportsHandTrackingMesh;
    uint32_t           maxHandMeshIndexCount;
    uint32_t           maxHandMeshVertexCount;
} XrSystemHandTrackingMeshPropertiesMSFT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `supportsHandTrackingMesh` is an `XrBool32`, indicating if current system is capable of hand tracking mesh input.
- `maxHandMeshIndexCount` is a `uint32_t` returns the maximum count of indices that will be returned from the hand tracker.
- `maxHandMeshVertexCount` is a `uint32_t` returns the maximum count of vertices that will be returned from the hand tracker.

If a runtime returns `XR_FALSE` for `supportsHandTrackingMesh`, the system does not support hand tracking mesh input, and therefore **must** return `XR_ERROR_FEATURE_UNSUPPORTED` from xrCreateHandMeshSpaceMSFT and xrUpdateHandMeshMSFT. The application **should** avoid using hand mesh functionality when `supportsHandTrackingMesh` is `XR_FALSE`.

If a runtime returns `XR_TRUE` for `supportsHandTrackingMesh`, the system supports hand tracking mesh input. In this case, the runtime **must** return a positive number for `maxHandMeshIndexCount` and `maxHandMeshVertexCount`. An application **should** use `maxHandMeshIndexCount` and `maxHandMeshVertexCount` to preallocate hand mesh buffers and reuse them in their render loop when calling xrUpdateHandMeshMSFT every frame.

Valid Usage (Implicit)

- The `XR_MSFT_hand_tracking_mesh` extension **must** be enabled prior to using XrSystemHandTrackingMeshPropertiesMSFT
- `type` **must** be `XR_TYPE_SYSTEM_HAND_TRACKING_MESH_PROPERTIES_MSFT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

Obtain a hand tracker handle

An application first creates an XrHandTrackerEXT handle using the xrCreateHandTrackerEXT function for each hand. The application can also reuse the same XrHandTrackerEXT handle previously created for the hand joint tracking. When doing so, the hand mesh input is always in sync with hand joints input with the same XrHandTrackerEXT handle.

Create a hand mesh space

The application creates a hand mesh space using function xrCreateHandMeshSpaceMSFT. The position and normal of hand mesh vertices will be represented in this space.

```
XrResult xrCreateHandMeshSpaceMSFT(
    XrHandTrackerEXT                            handTracker,
    const XrHandMeshSpaceCreateInfoMSFT*        createInfo,
    XrSpace*                                    space);
```

Parameter Descriptions

- `handTracker` is an XrHandTrackerEXT handle previously created with the xrCreateHandTrackerEXT function.
- `createInfo` is the XrHandMeshSpaceCreateInfoMSFT used to specify the hand mesh space.
- `space` is the returned XrSpace handle of the new hand mesh space.

A hand mesh space location is specified by runtime preference to effectively represent hand mesh vertices without unnecessary transformations. For example, an optical hand tracking system **can** define the hand mesh space origin at the depth camera's optical center.

An application should create separate hand mesh space handles for each hand to retrieve the corresponding hand mesh data. The runtime **may** use the lifetime of this hand mesh space handle to manage the underlying device resources. Therefore, the application **should** destroy the hand mesh handle after it is finished using the hand mesh.

The hand mesh space can be related to other spaces in the session, such as view reference space, or grip action space from the */interaction_profiles/khr/simple_controller* interaction profile. The hand mesh space may be not locatable when the hand is outside of the tracking range, or if focus is removed from the application. In these cases, the runtime **must** not set the XR_SPACE_LOCATION_POSITION_VALID_BIT and XR_SPACE_LOCATION_ORIENTATION_VALID_BIT bits on calls to xrLocateSpace with the hand mesh space, and the application **should** avoid using the returned poses or query for hand mesh data.

If the underlying XrHandTrackerEXT is destroyed, the runtime **must** continue to support xrLocateSpace using the hand mesh space, and it **must** return space location with XR_SPACE_LOCATION_POSITION_VALID_BIT and XR_SPACE_LOCATION_ORIENTATION_VALID_BIT unset.

The application **may** create a mesh space for the reference hand by setting handPoseType to XR_HAND_POSE_TYPE_REFERENCE_OPEN_PALM_MSFT. Hand mesh spaces for the reference hand **must** only be locatable in reference to mesh spaces or joint spaces of the reference hand.

Valid Usage (Implicit)

- The XR_MSFT_hand_tracking_mesh extension **must** be enabled prior to calling xrCreateHandMeshSpaceMSFT
- handTracker **must** be a valid XrHandTrackerEXT handle
- createInfo **must** be a pointer to a valid XrHandMeshSpaceCreateInfoMSFT structure
- space **must** be a pointer to an XrSpace handle

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_POSE_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_FEATURE_UNSUPPORTED

```
typedef struct XrHandMeshSpaceCreateInfoMSFT {
    XrStructureType       type;
    const void*           next;
    XrHandPoseTypeMSFT    handPoseType;
    XrPosef               poseInHandMeshSpace;
} XrHandMeshSpaceCreateInfoMSFT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `handPoseType` is an XrHandPoseTypeMSFT used to specify the type of hand this mesh is tracking. Indices and vertices returned from xrUpdateHandMeshMSFT for a hand type will be relative to the corresponding space create with the same hand type.
- `poseInHandMeshSpace` is an XrPosef defining the position and orientation of the new space's origin within the natural reference frame of the hand mesh space.

Valid Usage (Implicit)

- The XR_MSFT_hand_tracking_mesh extension **must** be enabled prior to using XrHandMeshSpaceCreateInfoMSFT
- type **must** be XR_TYPE_HAND_MESH_SPACE_CREATE_INFO_MSFT
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- handPoseType **must** be a valid XrHandPoseTypeMSFT value

Locate the hand mesh

The application **can** use the xrUpdateHandMeshMSFT function to retrieve the hand mesh at a given timestamp. The hand mesh's vertices position and normal are represented in the hand mesh space created by xrCreateHandMeshSpaceMSFT with a same XrHandTrackerEXT.

```
XrResult xrUpdateHandMeshMSFT(
    XrHandTrackerEXT                            handTracker,
    const XrHandMeshUpdateInfoMSFT*             updateInfo,
    XrHandMeshMSFT*                             handMesh);
```

Parameter Descriptions

- handTracker is an XrHandTrackerEXT handle previously created with xrCreateHandTrackerEXT.
- updateInfo is a XrHandMeshUpdateInfoMSFT which contains information to query the hand mesh.
- handMesh is an XrHandMeshMSFT structure to receive the updates of hand mesh data.

The application **should** preallocate the index buffer and vertex buffer in XrHandMeshMSFT using the maxHandMeshIndexCount and maxHandMeshVertexCount from the XrSystemHandTrackingMeshPropertiesMSFT returned from the xrGetSystemProperties function.

The application **should** preallocate the XrHandMeshMSFT structure and reuse it for each frame so as to reduce the copies of data when underlying tracking data is not changed. The application should use indexBufferChanged and vertexBufferChanged in XrHandMeshMSFT to detect changes and avoid unnecessary data processing when there is no changes.

Valid Usage (Implicit)

- The XR_MSFT_hand_tracking_mesh extension **must** be enabled prior to calling xrUpdateHandMeshMSFT
- handTracker **must** be a valid XrHandTrackerEXT handle
- updateInfo **must** be a pointer to a valid XrHandMeshUpdateInfoMSFT structure
- handMesh **must** be a pointer to an XrHandMeshMSFT structure

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_TIME_INVALID
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_SIZE_INSUFFICIENT
- XR_ERROR_FEATURE_UNSUPPORTED

A XrHandMeshUpdateInfoMSFT describes the information to update a hand mesh.

```
typedef struct XrHandMeshUpdateInfoMSFT {
    XrStructureType       type;
    const void*           next;
    XrTime                time;
    XrHandPoseTypeMSFT    handPoseType;
} XrHandMeshUpdateInfoMSFT;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- time is the XrTime that describes the time for which the application wishes to query the hand mesh state.
- handPoseType is an XrHandPoseTypeMSFT which describes the type of hand pose of the hand mesh to update.

A runtime **may** not maintain a full history of hand mesh data, therefore the returned XrHandMeshMSFT might return data that's not exactly corresponding to the time input. If the runtime cannot return any tracking data for the given time at all, it **must** set isActive to XR_FALSE for the call to xrUpdateHandMeshMSFT. Otherwise, if the runtime returns isActive as XR_TRUE, the data in XrHandMeshMSFT must be valid to use.

An application can choose different handPoseType values to query the hand mesh data. The returned hand mesh **must** be consistent to the hand joint space location on the same XrHandTrackerEXT when using the same XrHandPoseTypeMSFT.

Valid Usage (Implicit)

- The XR_MSFT_hand_tracking_mesh extension **must** be enabled prior to using XrHandMeshUpdateInfoMSFT
- type **must** be XR_TYPE_HAND_MESH_UPDATE_INFO_MSFT
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- handPoseType **must** be a valid XrHandPoseTypeMSFT value

A XrHandMeshMSFT structure contains data and buffers to receive updates of hand mesh tracking data from xrUpdateHandMeshMSFT function.

```
typedef struct XrHandMeshMSFT {
    XrStructureType              type;
    void*                        next;
    XrBool32                     isActive;
    XrBool32                     indexBufferChanged;
    XrBool32                     vertexBufferChanged;
    XrHandMeshIndexBufferMSFT    indexBuffer;
    XrHandMeshVertexBufferMSFT   vertexBuffer;
} XrHandMeshMSFT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `isActive` is an `XrBool32` indicating if the current hand tracker is active.
- `indexBufferChanged` is an `XrBool32` indicating if the `indexBuffer` content was changed during the update.
- `vertexBufferChanged` is an `XrBool32` indicating if the `vertexBuffer` content was changed during the update.
- `indexBuffer` is an XrHandMeshIndexBufferMSFT returns the index buffer of the tracked hand mesh.
- `vertexBuffer` is an XrHandMeshVertexBufferMSFT returns the vertex buffer of the tracked hand mesh.

When the returned `isActive` value is `XR_FALSE`, the runtime indicates the hand is not actively tracked, for example, the hand is outside of sensor's range, or the input focus is taken away from the application. When the runtime returns `XR_FALSE` to `isActive`, it **must** set `indexBufferChanged` and `vertexBufferChanged` to `XR_FALSE`, and **must** not change the content in `indexBuffer` or `vertexBuffer`,

When the returned `isActive` value is `XR_TRUE`, the hand tracking mesh represented in `indexBuffer` and `vertexBuffer` are updated to the latest data of the `time` given to the xrUpdateHandMeshMSFT function. The runtime **must** set `indexBufferChanged` and `vertexBufferChanged` to reflect whether the index or vertex buffer's content are changed during the update. In this way, the application can easily avoid unnecessary processing of buffers when there's no new data.

The hand mesh is represented in triangle lists and each triangle's vertices are in counter-clockwise order when looking from outside of the hand. When hand tracking is active, i.e. when `isActive` is returned as `XR_TRUE`, the returned `indexBuffer.indexCountOutput` value **must** be positive and multiple of 3, and `vertexBuffer.vertexCountOutput` value **must** be equal to or larger than 3.

Valid Usage (Implicit)

- The XR_MSFT_hand_tracking_mesh extension **must** be enabled prior to using XrHandMeshMSFT
- type **must** be XR_TYPE_HAND_MESH_MSFT
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- indexBuffer **must** be a valid XrHandMeshIndexBufferMSFT structure
- vertexBuffer **must** be a valid XrHandMeshVertexBufferMSFT structure

A XrHandMeshIndexBufferMSFT structure includes an array of indices describing the triangle list of a hand mesh.

```
typedef struct XrHandMeshIndexBufferMSFT {
    uint32_t     indexBufferKey;
    uint32_t     indexCapacityInput;
    uint32_t     indexCountOutput;
    uint32_t*    indices;
} XrHandMeshIndexBufferMSFT;
```

Member Descriptions

- indexBufferKey is a uint32_t serving as the key of the returned index buffer content or 0 to indicate a request to retrieve the latest indices regardless of existing content in indices.
- indexCapacityInput is a positive uint32_t describes the capacity of the indices array.
- indexCountOutput is a uint32_t returned by the runtime with the count of indices written in indices.
- indices is an array of indices filled in by the runtime, specifying the indices of the triangles list in the vertex buffer.

An application **should** preallocate the indices array using the maxHandMeshIndexCount in XrSystemHandTrackingMeshPropertiesMSFT returned from xrGetSystemProperties. In this way, the application can avoid possible insufficient buffer sizees for each query, and therefore avoid reallocating memory each frame.

The input indexCapacityInput **must** not be 0, and indices **must** not be NULL, or else the runtime **must** return XR_ERROR_VALIDATION_FAILURE on calls to the xrUpdateHandMeshMSFT function.

If the input indexCapacityInput is not sufficient to contain all output indices, the runtime **must** return XR_ERROR_SIZE_INSUFFICIENT on calls to xrUpdateHandMeshMSFT, not change the content in indexBufferKey and indices, and return 0 for indexCountOutput.

If the input indexCapacityInput is equal to or larger than the maxHandMeshIndexCount in XrSystemHandTrackingMeshPropertiesMSFT returned from xrGetSystemProperties, the runtime **must** not return XR_ERROR_SIZE_INSUFFICIENT error on xrUpdateHandMeshMSFT because of insufficient index buffer size.

If the input indexBufferKey is 0, the capacity of indices array is sufficient, and hand mesh tracking is active, the runtime **must** return the latest non-zero indexBufferKey, and fill in indexCountOutput and indices.

If the input indexBufferKey is not 0, the runtime **can** either return without changing indexCountOutput or content in indices, and return XR_FALSE for indexBufferChanged indicating the indices are not changed; or return a new non-zero indexBufferKey and fill in latest data in indexCountOutput and indices, and return XR_TRUE for indexBufferChanged indicating the indices are updated to a newer version.

An application **can** keep the XrHandMeshIndexBufferMSFT structure for each frame in a frame loop and use the returned indexBufferKey to identify different triangle list topology described in indices. The application can therefore avoid unnecessary processing of indices, such as coping them to GPU memory.

The runtime **must** return the same indexBufferKey for the same XrHandTrackerEXT at a given time, regardless of the input XrHandPoseTypeMSFT in XrHandMeshUpdateInfoMSFT. This ensures the index buffer has the same mesh topology and allows the application to reason about vertices across different hand pose types. For example, the application **can** build a procedure to perform UV mapping on vertices of a hand mesh using XR_HAND_POSE_TYPE_REFERENCE_OPEN_PALM_MSFT, and apply the resultant UV data on vertices to the mesh returned from the same hand tracker using XR_HAND_POSE_TYPE_TRACKED_MSFT.

Valid Usage (Implicit)

- The XR_MSFT_hand_tracking_mesh extension **must** be enabled prior to using XrHandMeshIndexBufferMSFT
- indices **must** be a pointer to an array of indexCapacityInput uint32_t values
- The indexCapacityInput parameter **must** be greater than 0

A XrHandMeshVertexBufferMSFT structure includes an array of vertices of the hand mesh represented in the hand mesh space.

```
typedef struct XrHandMeshVertexBufferMSFT {
    XrTime                   vertexUpdateTime;
    uint32_t                 vertexCapacityInput;
    uint32_t                 vertexCountOutput;
    XrHandMeshVertexMSFT*    vertices;
} XrHandMeshVertexBufferMSFT;
```

Member Descriptions

- `vertexUpdateTime` is an `XrTime` representing the time when the runtime receives the vertex buffer content or 0 to indicate a request to retrieve latest vertices regardless of existing content in `vertices`.
- `vertexCapacityInput` is a positive `uint32_t` describes the capacity of the `vertices` array.
- `vertexCountOutput` is a `uint32_t` filled in by the runtime with the count of vertices written in `vertices`.
- `vertices` is an array of XrHandMeshVertexMSFT filled in by the runtime, specifying the vertices of the hand mesh including the position and normal vector in the hand mesh space.

An application **should** preallocate the vertices array using the `maxHandMeshVertexCount` in XrSystemHandTrackingMeshPropertiesMSFT returned from xrGetSystemProperties. In this way, the application can avoid possible insufficient buffer sizes for each query, and therefore avoid reallocating memory each frame.

The input `vertexCapacityInput` **must** not be 0, and `vertices` **must** not be `NULL`, or else the runtime **must** return `XR_ERROR_VALIDATION_FAILURE` on calls to the xrUpdateHandMeshMSFT function.

If the input `vertexCapacityInput` is not sufficient to contain all output vertices, the runtime **must** return `XR_ERROR_SIZE_INSUFFICIENT` on calls to the xrUpdateHandMeshMSFT, do not change content in `vertexUpdateTime` and `vertices`, and return 0 for `vertexCountOutput`.

If the input `vertexCapacityInput` is equal to or larger than the `maxHandMeshVertexCount` in XrSystemHandTrackingMeshPropertiesMSFT returned from xrGetSystemProperties, the runtime **must** not return `XR_ERROR_SIZE_INSUFFICIENT` on calls to the xrUpdateHandMeshMSFT because of insufficient vertex buffer size.

If the input `vertexUpdateTime` is 0, and the capacity of the vertices array is sufficient, and hand mesh tracking is active, the runtime **must** return the latest non-zero `vertexUpdateTime`, and fill in the `vertexCountOutput` and `vertices` fields.

If the input `vertexUpdateTime` is not 0, the runtime **can** either return without changing `vertexCountOutput` or the content in `vertices`, and return `XR_FALSE` for `vertexBufferChanged` indicating the vertices are not changed; or return a new non-zero `vertexUpdateTime` and fill in latest data in `vertexCountOutput` and `vertices` and return `XR_TRUE` for `vertexBufferChanged` indicating the vertices are updated to a newer version.

An application **can** keep the XrHandMeshVertexBufferMSFT structure for each frame in frame loop and use the returned vertexUpdateTime to detect the changes of the content in vertices. The application can therefore avoid unnecessary processing of vertices, such as coping them to GPU memory.

Valid Usage (Implicit)

- The XR_MSFT_hand_tracking_mesh extension **must** be enabled prior to using XrHandMeshVertexBufferMSFT
- vertices **must** be a pointer to an array of vertexCapacityInput XrHandMeshVertexMSFT structures
- The vertexCapacityInput parameter **must** be greater than 0

Each XrHandMeshVertexMSFT includes the position and normal of a vertex of a hand mesh.

```
typedef struct XrHandMeshVertexMSFT {
    XrVector3f    position;
    XrVector3f    normal;
} XrHandMeshVertexMSFT;
```

Member Descriptions

- position is an XrVector3f structure representing the position of the vertex in the hand mesh space, measured in meters.
- normal is an XrVector3f structure representing the unweighted normal of the triangle surface at the vertex as a unit vector in hand mesh space.

Valid Usage (Implicit)

- The XR_MSFT_hand_tracking_mesh extension **must** be enabled prior to using XrHandMeshVertexMSFT

Example code for hand mesh tracking

Following example code demos preallocating hand mesh buffers and updating the hand mesh in rendering loop

```
XrInstance instance; // previously initialized
XrSystemId systemId; // previously initialized
XrSession session;   // previously initialized

// Inspect hand tracking mesh system properties
XrSystemHandTrackingMeshPropertiesMSFT
handMeshSystemProperties{XR_TYPE_SYSTEM_HAND_TRACKING_MESH_PROPERTIES_MSFT};
XrSystemProperties systemProperties{XR_TYPE_SYSTEM_PROPERTIES,
&handMeshSystemProperties};
CHK_XR(xrGetSystemProperties(instance, systemId, &systemProperties));
if (!handMeshSystemProperties.supportsHandTrackingMesh) {
    // the system does not support hand mesh tracking
    return;
}

// Get function pointer for xrCreateHandTrackerEXT
PFN_xrCreateHandTrackerEXT pfnCreateHandTrackerEXT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrCreateHandTrackerEXT",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                             &pfnCreateHandTrackerEXT)));

// Create a tracker for left hand.
XrHandTrackerEXT leftHandTracker{};
{
    XrHandTrackerCreateInfoEXT createInfo{XR_TYPE_HAND_TRACKER_CREATE_INFO_EXT};
    createInfo.hand = XR_HAND_LEFT_EXT;
    createInfo.handJointSet = XR_HAND_JOINT_SET_DEFAULT_EXT;
    CHK_XR(pfnCreateHandTrackerEXT(session, &createInfo, &leftHandTracker));
}

// Get function pointer for xrCreateHandMeshSpaceMSFT
PFN_xrCreateHandMeshSpaceMSFT pfnCreateHandMeshSpaceMSFT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrCreateHandMeshSpaceMSFT",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                             &pfnCreateHandMeshSpaceMSFT)));

// Create the hand mesh spaces
XrSpace leftHandMeshSpace{};
{
    XrHandMeshSpaceCreateInfoMSFT createInfo{XR_TYPE_HAND_MESH_SPACE_CREATE_INFO_MSFT};
    createInfo.poseInHandMeshSpace = {{0, 0, 0, 1}, {0, 0, 0}};
    CHK_XR(pfnCreateHandMeshSpaceMSFT(leftHandTracker, &createInfo, &leftHandMeshSpace));
}

// Preallocate buffers for hand mesh indices and vertices
std::vector<uint32_t> handMeshIndices(handMeshSystemProperties.maxHandMeshIndexCount);
std::vector<XrHandMeshVertexMSFT>
handMeshVertices(handMeshSystemProperties.maxHandMeshVertexCount);
```

```
XrHandMeshMSFT leftHandMesh{XR_TYPE_HAND_MESH_MSFT};
leftHandMesh.indexBuffer.indexCapacityInput = (uint32_t)handMeshIndices.size();
leftHandMesh.indexBuffer.indices = handMeshIndices.data();
leftHandMesh.vertexBuffer.vertexCapacityInput = (uint32_t)handMeshVertices.size();
leftHandMesh.vertexBuffer.vertices = handMeshVertices.data();

// Get function pointer for xrUpdateHandMeshMSFT
PFN_xrUpdateHandMeshMSFT pfnUpdateHandMeshMSFT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrUpdateHandMeshMSFT",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                             &pfnUpdateHandMeshMSFT)));
while(1){
    // ...
    // For every frame in frame loop
    // ...
    XrFrameState frameState;    // previously returned from xrWaitFrame
    const XrTime time = frameState.predictedDisplayTime;

    XrHandMeshUpdateInfoMSFT updateInfo{XR_TYPE_HAND_MESH_UPDATE_INFO_MSFT};
    updateInfo.time = time;
    CHK_XR(pfnUpdateHandMeshMSFT(leftHandTracker, &updateInfo, &leftHandMesh));
    if (!leftHandMesh.isActive) {
        // Hand input is not focused or user's hand is out of tracking range.
        // Do not process or render hand mesh.
    } else {
        if (leftHandMesh.indexBufferChanged) {
            // Process indices in indexBuffer.indices
        }

        if (leftHandMesh.vertexBufferChanged) {
            // Process vertices in vertexBuffer.vertices and leftHandMeshSpace
        }
    }
}
```

Get hand reference poses

By default, a XrHandTrackerEXT tracks a default hand pose type, that is to provide best fidelity to the user's actual hand motion. This is the same with `XR_HAND_POSE_TYPE_TRACKED_MSFT` (i.e. value 0) in a chained XrHandPoseTypeInfoMSFT structure to the `next` pointer of XrHandTrackerCreateInfoEXT when calling xrCreateHandTrackerEXT.

Some hand mesh visualizations may require an initial analysis or processing of the hand mesh relative to the joints of the hand. For example, a hand visualization may generate a UV mapping for the hand mesh vertices by raycasting outward from key joints against the mesh to find key vertices.

To avoid biasing such static analysis with the arbitrary tracked hand pose, an application **can** instead create a different XrHandTrackerEXT handle with a reference hand pose type when calling xrCreateHandTrackerEXT. This will instruct the runtime to provide a reference hand pose that is better suited for such static analysis.

An application can chain an XrHandPoseTypeInfoMSFT structure to the XrHandTrackerCreateInfoEXT::`next` pointer when calling xrCreateHandTrackerEXT to indicate the hand tracker to return the hand pose of specific XrHandPoseTypeMSFT.

```
typedef struct XrHandPoseTypeInfoMSFT {
    XrStructureType       type;
    const void*           next;
    XrHandPoseTypeMSFT    handPoseType;
} XrHandPoseTypeInfoMSFT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `handPoseType` is an XrHandPoseTypeMSFT that describes the type of hand pose of the hand tracking.

Valid Usage (Implicit)

- The `XR_MSFT_hand_tracking_mesh` extension **must** be enabled prior to using XrHandPoseTypeInfoMSFT
- `type` **must** be `XR_TYPE_HAND_POSE_TYPE_INFO_MSFT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `handPoseType` **must** be a valid XrHandPoseTypeMSFT value

The XrHandPoseTypeMSFT describes the type of input hand pose from XrHandTrackerEXT.

```
typedef enum XrHandPoseTypeMSFT {
    XR_HAND_POSE_TYPE_TRACKED_MSFT = 0,
    XR_HAND_POSE_TYPE_REFERENCE_OPEN_PALM_MSFT = 1,
    XR_HAND_POSE_TYPE_MAX_ENUM_MSFT = 0x7FFFFFFF
} XrHandPoseTypeMSFT;
```

Enumerant Descriptions

- XR_HAND_POSE_TYPE_TRACKED_MSFT represents a hand pose provided by actual tracking of the user's hand.
- XR_HAND_POSE_TYPE_REFERENCE_OPEN_PALM_MSFT represents a stable reference hand pose in a relaxed open hand shape.

The XR_HAND_POSE_TYPE_TRACKED_MSFT input provides best fidelity to the user's actual hand motion. When the hand tracking input requires the user to be holding a controller in their hand, the hand tracking input will appear as the user virtually holding the controller. This input can be used to render the hand shape together with the controller in hand.

The XR_HAND_POSE_TYPE_REFERENCE_OPEN_PALM_MSFT input does not move with the user's actual hand. Through this reference hand pose, an application **can** get a stable hand joint and mesh that has the same mesh topology as the tracked hand mesh using the same XrHandTrackerEXT, so that the application can apply the data computed from a reference hand pose to the corresponding tracked hand.

Although a reference hand pose does not move with user's hand motion, the bone length and hand thickness **may** be updated, for example when tracking result refines, or a different user's hand is detected. The application **should** update reference hand joints and meshes when the tracked mesh's indexBufferKey is changed or when the isActive value returned from xrUpdateHandMeshMSFT changes from XR_FALSE to XR_TRUE. It can use the returned indexBufferKey and vertexUpdateTime from xrUpdateHandMeshMSFT to avoid unnecessary CPU or GPU work to process the neutral hand inputs.

Example code for reference hand mesh update

The following example code demonstrates detecting reference hand mesh changes and retrieving data for processing.

```
XrInstance instance;                // previously initialized
XrSession session;                  // previously initialized
XrHandTrackerEXT handTracker;       // previously initialized with handJointSet set to
XR_HAND_JOINT_SET_DEFAULT_MSFT
XrSpace handMeshReferenceSpace;     // previously initialized with handPoseType set to
XR_HAND_POSE_TYPE_REFERENCE_OPEN_PALM_MSFT
XrHandMeshMSFT referenceHandMesh;   // previously initialized with preallocated buffers
```

```
// Get function pointer for xrUpdateHandMeshMSFT
PFN_xrUpdateHandMeshMSFT pfnUpdateHandMeshMSFT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrUpdateHandMeshMSFT",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                             &pfnUpdateHandMeshMSFT)));

// Get function pointer for xrCreateHandTrackerEXT
PFN_xrCreateHandTrackerEXT pfnCreateHandTrackerEXT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrCreateHandTrackerEXT",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                             &pfnCreateHandTrackerEXT)));

// Get function pointer for xrLocateHandJointsEXT
PFN_xrLocateHandJointsEXT pfnLocateHandJointsEXT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrLocateHandJointsEXT",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                             &pfnLocateHandJointsEXT)));
while(1){
    // ...
    // For every frame in frame loop
    // ...
    XrFrameState frameState;    // previously returned from xrWaitFrame
    const XrTime time = frameState.predictedDisplayTime;

    XrHandMeshUpdateInfoMSFT updateInfo{XR_TYPE_HAND_MESH_UPDATE_INFO_MSFT};
    updateInfo.time = time;
    updateInfo.handPoseType = XR_HAND_POSE_TYPE_REFERENCE_OPEN_PALM_MSFT;
    CHK_XR(pfnUpdateHandMeshMSFT(handTracker, &updateInfo, &referenceHandMesh));

    // Detect if reference hand mesh is changed.
    if (referenceHandMesh.indexBufferChanged || referenceHandMesh.vertexBufferChanged) {

        // Query the joint location using "open palm" reference hand pose.
        XrHandPoseTypeInfoMSFT handPoseTypeInfo{XR_TYPE_HAND_POSE_TYPE_INFO_MSFT};
        handPoseTypeInfo.handPoseType = XR_HAND_POSE_TYPE_REFERENCE_OPEN_PALM_MSFT;

        XrHandTrackerCreateInfoEXT createInfo{XR_TYPE_HAND_TRACKER_CREATE_INFO_EXT};
        createInfo.hand = XR_HAND_LEFT_EXT;
        createInfo.handJointSet = XR_HAND_JOINT_SET_DEFAULT_EXT;
        createInfo.next = &handPoseTypeInfo;

        XrHandTrackerEXT referenceHandTracker;
        CHK_XR(pfnCreateHandTrackerEXT(session, &createInfo, &referenceHandTracker));

        XrHandJointsLocateInfoEXT locateInfo{XR_TYPE_HAND_JOINTS_LOCATE_INFO_EXT};
        locateInfo.next = &handPoseTypeInfo;
        locateInfo.baseSpace = handMeshReferenceSpace;  // Query joint location relative
```

```
to hand mesh reference space
        locateInfo.time = time;

        std::array<XrHandJointLocationEXT, XR_HAND_JOINT_COUNT_EXT> jointLocations;
        XrHandJointLocationsEXT locations{XR_TYPE_HAND_JOINT_LOCATIONS_EXT};
        locations.jointCount = jointLocations.size();
        locations.jointLocations = jointLocations.data();

        CHK_XR(pfnLocateHandJointsEXT(referenceHandTracker, &locateInfo, &locations));

        // Generate UV map using tip/wrist location and referenceHandMesh.vertexBuffer
        // For example, gradually changes color from the tip of the hand to wrist.
    }
}
```

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- XR_TYPE_HAND_MESH_SPACE_CREATE_INFO_MSFT
- XR_TYPE_HAND_MESH_UPDATE_INFO_MSFT
- XR_TYPE_HAND_MESH_MSFT
- XR_TYPE_SYSTEM_HAND_TRACKING_MESH_PROPERTIES_MSFT
- XR_TYPE_HAND_POSE_TYPE_INFO_MSFT

New Enums

- XrHandPoseTypeMSFT

New Structures

- XrHandMeshSpaceCreateInfoMSFT
- XrHandMeshUpdateInfoMSFT
- XrHandMeshMSFT
- XrHandMeshIndexBufferMSFT
- XrHandMeshVertexBufferMSFT
- XrHandMeshVertexMSFT
- XrSystemHandTrackingMeshPropertiesMSFT
- XrHandPoseTypeInfoMSFT

New Functions

- xrCreateHandMeshSpaceMSFT
- xrUpdateHandMeshMSFT

Issues

Version History

- Revision 1, 2019-09-20 (Yin LI)
  - Initial extension description
- Revision 2, 2020-04-20 (Yin LI)
  - Change joint spaces to locate joints function.
- Revision 3, 2021-04-13 (Ryan Pavlik, Collabora)
  - Correctly show function pointer retrieval in sample code

12.47. XR_MSFT_holographic_window_attachment

Name String

`XR_MSFT_holographic_window_attachment`

Extension Type

Instance extension

Registered Extension Number

64

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Contributors

Bryce Hutchings, Microsoft
Yin Li, Microsoft
Alex Turner, Microsoft

Overview

This extension enables the runtime to attach to app-provided HolographicSpace and CoreWindow WinRT objects when an XrSession is created. Applications may use this extension to create and control the CoreWindow/App View objects, allowing the app to subscribe to keyboard input events and react to activation event arguments. These events and data would otherwise be inaccessible if the application simply managed the app state and lifetime exclusively through the OpenXR API. This extension is only valid to use where an application can create a CoreWindow, such as UWP applications on the HoloLens.

The XrHolographicWindowAttachmentMSFT structure is defined as:

```
typedef struct XrHolographicWindowAttachmentMSFT {
    XrStructureType    type;
    const void*        next;
    IUnknown*          holographicSpace;
    IUnknown*          coreWindow;
} XrHolographicWindowAttachmentMSFT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `holographicSpace` is a pointer to a valid `Windows`.Graphics.Holographic.HolographicSpace.
- `coreWindow` is a pointer to a valid `Windows`.UI.Core.CoreWindow.

When creating a holographic window-backed XrSession, the application provides a pointer to an XrHolographicWindowAttachmentMSFT in the `next` chain of the XrSessionCreateInfo.

The session state of a holographic window-backed XrSession will only reach `XR_SESSION_STATE_VISIBLE` when the provided CoreWindow is made visible. If the CoreWindow is for a secondary app view, the application must programmatically request to make the CoreWindow visible (e.g. with `ApplicationViewSwitcher.TryShowAsStandaloneAsync` or `ApplicationViewSwitcher.SwitchAsync`).

The app **must** not call xrCreateSession while the specified CoreWindow thread is blocked, otherwise the call **may** deadlock.

Valid Usage (Implicit)

- The XR_MSFT_holographic_window_attachment extension **must** be enabled prior to using XrHolographicWindowAttachmentMSFT
- type **must** be XR_TYPE_HOLOGRAPHIC_WINDOW_ATTACHMENT_MSFT
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- holographicSpace **must** be a pointer to an IUnknown value
- coreWindow **must** be a pointer to an IUnknown value

12.47.1. Sample code

Following example demos the usage of holographic window attachment and use the attached CoreWindow to receive keyboard input, use CoreTextEditContext to handle text typing experience, and use IActivatedEventArgs to handle protocol launching arguments.

```
struct AppView : implements<AppView, IFrameworkView> {
    void Initialize(CoreApplicationView const& applicationView) {
        applicationView.Activated({this, &AppView::OnActivated});
    }

    void Load(winrt::hstring const& entryPoint) {
    }

    void Uninitialize() {
    }

    void Run() {
        // Creating a HolographicSpace before activating the CoreWindow to make it a
holographic window
        CoreWindow window = CoreWindow::GetForCurrentThread();
        HolographicSpace holographicSpace =
Windows::Graphics::Holographic::HolographicSpace::CreateForCoreWindow(window);
        window.Activate();

        // [xrCreateInstance, xrGetSystem, and create a graphics binding]

        XrHolographicWindowAttachmentMSFT
holographicWindowAttachment{XR_TYPE_ATTACHED_CORE_WINDOW_MSFT};
        holographicWindowAttachment.next = &graphicsBinding;
        holographicWindowAttachment.coreWindow = window.as<IUnknown>().get();
        holographicWindowAttachment.holographicSpace =
holographicSpace.as<IUnknown>().get();
```

```
        XrSessionCreateInfo sessionCreateInfo{XR_TYPE_SESSION_CREATE_INFO};
        sessionCreateInfo.next = &holographicWindowAttachment;
        sessionCreateInfo.systemId = systemId;

        XrSession session;
        CHECK_XRCMD(xrCreateSession(instance, &sessionCreateInfo, &session));

        while (!m_windowClosed) {

window.Dispatcher().ProcessEvents(CoreProcessEventsOption::ProcessAllIfPresent);

            // [OpenXR calls: Poll events, sync actions, render, and submit frames].
        }
    }

    void SetWindow(CoreWindow const& window) {
        window.Closed({this, &AppView::OnWindowClosed});
        window.KeyDown({this, &AppView::OnKeyDown});

        // This sample customizes the text input pane with manual display policy and
email address scope.
        windows::CoreTextServicesManager manager =
windows::CoreTextServicesManager::GetForCurrentView();
        windows::CoreTextEditContext editingContext = manager.CreateEditContext();

editingContext.InputPaneDisplayPolicy(windows::CoreTextInputPaneDisplayPolicy::Manual);
        editingContext.InputScope(windows::CoreTextInputScope::EmailAddress);
    }

    void OnWindowClosed(CoreWindow const& sender, CoreWindowEventArgs const& args) {
        m_windowClosed = true;
    }

    void OnKeyDown(CoreWindow const& sender, KeyEventArgs const& args) {
        // [Process key down]
    }

    void OnActivated(CoreApplicationView const&, IActivatedEventArgs const& args) {
        if (args.Kind() == windows::ActivationKind::Protocol) {
            auto eventArgs{args.as<windows::ProtocolActivatedEventArgs>()};
            // Use the protocol activation parameters in eventArgs.Uri();
        }

        // Inspecting whether the application is launched from within holographic shell
or from desktop.
        if (windows::HolographicApplicationPreview::IsHolographicActivation(args)) {
            // App activation is targeted at the holographic shell.
        } else {
```

```
            // App activation is targeted at the desktop.
        }

        // NOTE: CoreWindow is activated later after the HolographicSpace has been
created.
    }

    bool m_windowClosed{false};
};

struct AppViewSource : winrt::implements<AppViewSource, IFrameworkViewSource> {
    windows::IFrameworkView CreateView() {
        return winrt::make<AppView>();
    }
};

int __stdcall wWinMain(HINSTANCE, HINSTANCE, PWSTR, int) {
    CoreApplication::Run(make<AppViewSource>());
}
```

Version History

- Revision 1, 2020-05-18 (Bryce Hutchings)
  - Initial extension description

12.48. XR_MSFT_perception_anchor_interop

Name String

`XR_MSFT_perception_anchor_interop`

Extension Type

Instance extension

Registered Extension Number

57

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0
- Requires `XR_MSFT_spatial_anchor`

Last Modified Date

2020-06-16

IP Status

No known IP claims.

Contributors

Lachlan Ford, Microsoft
Bryce Hutchings, Microsoft
Yin Li, Microsoft

Overview

This extension supports conversion between XrSpatialAnchorMSFT and Windows.Perception.Spatial.SpatialAnchor. An application **can** use this extension to persist spatial anchors on the Windows device through SpatialAnchorStore or transfer spatial anchors between devices through SpatialAnchorTransferManager.

The xrCreateSpatialAnchorFromPerceptionAnchorMSFT function creates a XrSpatialAnchorMSFT handle from an `IUnknown` pointer to Windows.Perception.Spatial.SpatialAnchor.

```
XrResult xrCreateSpatialAnchorFromPerceptionAnchorMSFT(
    XrSession                                   session,
    IUnknown*                                   perceptionAnchor,
    XrSpatialAnchorMSFT*                        anchor);
```

Parameter Descriptions

- `session` is the specified XrSession.
- `perceptionAnchor` is an IUnknown pointer to a Windows.Perception.Spatial.SpatialAnchor object.
- `anchor` is a pointer to XrSpatialAnchorMSFT to receive the returned anchor handle.

The input `perceptionAnchor` **must** support successful `QueryInterface` to Windows.Perception.Spatial.SpatialAnchor , otherwise the runtime **must** return `XR_ERROR_VALIDATION_FAILURE`.

If the function successfully returned, the output `anchor` **must** be a valid handle. This also increments the refcount of the `perceptionAnchor` object.

When application is done with the `anchor` handle, it **can** be destroyed using xrDestroySpatialAnchorMSFT function. This also decrements the refcount of underlying windows perception anchor object.

Valid Usage (Implicit)

- The XR_MSFT_perception_anchor_interop extension **must** be enabled prior to calling xrCreateSpatialAnchorFromPerceptionAnchorMSFT
- session **must** be a valid XrSession handle
- perceptionAnchor **must** be a pointer to an IUnknown value
- anchor **must** be a pointer to an XrSpatialAnchorMSFT handle

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_SESSION_LOST
- XR_SESSION_LOSS_PENDING

The xrTryGetPerceptionAnchorFromSpatialAnchorMSFT function converts a XrSpatialAnchorMSFT handle into an IUnknown pointer to Windows.Perception.Spatial.SpatialAnchor.

```
XrResult xrTryGetPerceptionAnchorFromSpatialAnchorMSFT(
    XrSession                                   session,
    XrSpatialAnchorMSFT                         anchor,
    IUnknown**                                  perceptionAnchor);
```

Parameter Descriptions

- `session` is the specified XrSession.
- `anchor` is a valid XrSpatialAnchorMSFT handle.
- `perceptionAnchor` is a valid pointer to IUnknown pointer to receive the output Windows.Perception.Spatial.SpatialAnchor object.

If the runtime can convert the `anchor` to a Windows.Perception.Spatial.SpatialAnchor object, this function **must** return `XR_SUCCESS`, and the output `IUnknown` in the pointer of `perceptionAnchor` **must** be not `NULL`. This also increments the refcount of the object. The application **can** then use `QueryInterface` to get the pointer for Windows.Perception.Spatial.SpatialAnchor object. The application **should** release the COM pointer after done with the object, or attach it to a smart COM pointer such as `winrt::com_ptr`.

If the runtime cannot convert the `anchor` to a Windows.Perception.Spatial.SpatialAnchor object, the function **must** return `XR_SUCCESS`, and the output `IUnknown` in the pointer of `perceptionAnchor` **must** be `NULL`.

Valid Usage (Implicit)

- The `XR_MSFT_perception_anchor_interop` extension **must** be enabled prior to calling xrTryGetPerceptionAnchorFromSpatialAnchorMSFT
- `session` **must** be a valid XrSession handle
- `anchor` **must** be a valid XrSpatialAnchorMSFT handle
- `perceptionAnchor` **must** be a pointer to a pointer to an `IUnknown` value
- `anchor` **must** have been created, allocated, or retrieved from `session`

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_SESSION_LOST
- XR_SESSION_LOSS_PENDING

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

New Functions

xrCreateSpatialAnchorFromPerceptionAnchorMSFT

xrTryGetPerceptionAnchorFromSpatialAnchorMSFT

Issues

Version History

- Revision 1, 2020-06-16 (Yin Li)
  - Initial extension proposal

12.49. XR_MSFT_secondary_view_configuration

Name String

XR_MSFT_secondary_view_configuration

Extension Type

Instance extension

Registered Extension Number

54

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2020-05-02

IP Status

No known IP claims.

Contributors

Yin Li, Microsoft
Zonglin Wu, Microsoft
Alex Turner, Microsoft

12.49.1. Overview

This extension allows an application to enable support for one or more **secondary view configurations**. A secondary view configuration is a well-known set of views that the runtime can make active while a session is running. In a frame where a secondary view configuration is active, the application's single frame loop should additionally render into those active secondary views, sharing the frame waiting logic and update loop with the primary view configuration for that running session.

A proper secondary view configuration support includes following steps:

1. When calling xrCreateInstance, enable the XR_MSFT_secondary_view_configuration extension and the extension defines a concrete secondary view configuration type, for example, XR_MSFT_first_person_observer.
2. Inspect supported secondary view configurations using the xrEnumerateViewConfigurations function.
3. Enable supported secondary view configurations using the xrBeginSession function with an XrSecondaryViewConfigurationSessionBeginInfoMSFT chained extension structure.
4. Inspect if an enabled secondary view configuration is activated by the system or the user using the xrWaitFrame function with an XrSecondaryViewConfigurationFrameStateMSFT chained extension structure.
5. When a secondary view configuration is changed to active, get the latest view configuration

properties using the xrGetViewConfigurationProperties and xrEnumerateViewConfigurationViews functions.

6. Create the swapchain images for the active secondary view configuration using the xrCreateSwapchain function with an XrSecondaryViewConfigurationSwapchainCreateInfoMSFT chained extension structure using `recommendedImageRectWidth` and `recommendedImageRectHeight` in the corresponding XrViewConfigurationView structure returned from xrEnumerateViewConfigurationViews.
7. Locate the secondary view configuration views using the xrLocateViews function with the active secondary view configuration type.
8. Submit the composition layers using the swapchain images for an active secondary view configuration using the xrEndFrame function with the XrSecondaryViewConfigurationFrameEndInfoMSFT chained extension structure.

12.49.2. Enumerate supported secondary view configurations

The first step is for the application to inspect if a runtime supports certain secondary view configurations. The app uses the existing API xrEnumerateViewConfigurations for this.

For example, when the XR_MSFT_first_person_observer extension is enabled, the application will enumerate a view configuration of type `XR_VIEW_CONFIGURATION_TYPE_SECONDARY_MONO_FIRST_PERSON_OBSERVER_MSFT`, and can use this secondary view configuration type in later functions.

12.49.3. Secondary view configuration properties

The application can inspect the properties of a secondary view configuration through the existing xrGetViewConfigurationProperties, xrEnumerateViewConfigurationViews and xrEnumerateEnvironmentBlendModes functions using a supported secondary view configuration type.

The runtime **may** change the recommended properties, such as recommended image width or height, when the secondary view configuration becomes active. The application **should** use the latest recommended width and height when creating swapchain images and related resources for the active secondary view configuration.

When an application creates swapchain images for a secondary view configuration, it **can** chain a XrSecondaryViewConfigurationSwapchainCreateInfoMSFT structure to XrSwapchainCreateInfo when calling xrCreateSwapchain. This hints to the runtime that the created swapchain image will be submitted to the given secondary view configuration, allowing the runtime to make optimizations for such usage when there is opportunity.

```
typedef struct XrSecondaryViewConfigurationSwapchainCreateInfoMSFT {
    XrStructureType            type;
    const void*                next;
    XrViewConfigurationType    viewConfigurationType;
} XrSecondaryViewConfigurationSwapchainCreateInfoMSFT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `viewConfigurationType` is the secondary view configuration type the application is intending to use this swapchain for.

If this structure is not present in the XrSwapchainCreateInfo next chain when calling xrCreateSwapchain, the runtime **should** optimize the created swapchain for the primary view configuration of the session.

If the application submits a swapchain image created with one view configuration type to a composition layer for another view configuration, the runtime **may** need to copy the resource across view configurations. However, the runtime **must** correctly compose the image regardless which view configuration type was hinted when swapchain image was created.

Valid Usage (Implicit)

- The `XR_MSFT_secondary_view_configuration` extension **must** be enabled prior to using XrSecondaryViewConfigurationSwapchainCreateInfoMSFT
- `type` **must** be `XR_TYPE_SECONDARY_VIEW_CONFIGURATION_SWAPCHAIN_CREATE_INFO_MSFT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `viewConfigurationType` **must** be a valid XrViewConfigurationType value

12.49.4. Enable secondary view configuration

The application indicates to the runtime which secondary view configurations it can support by chaining an XrSecondaryViewConfigurationSessionBeginInfoMSFT structure to the XrSessionBeginInfo::`next` pointer when calling xrBeginSession.

The XrSecondaryViewConfigurationSessionBeginInfoMSFT structure is used by the application to indicate the list of secondary XrViewConfigurationType to enable for this session.

It is defined as:

```
typedef struct XrSecondaryViewConfigurationSessionBeginInfoMSFT {
    XrStructureType                  type;
    const void*                      next;
    uint32_t                         viewConfigurationCount;
    const XrViewConfigurationType*   enabledViewConfigurationTypes;
} XrSecondaryViewConfigurationSessionBeginInfoMSFT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `viewConfigurationCount` is the number of elements in `enabledViewConfigurationTypes`
- `enabledViewConfigurationTypes` is an array of enabled secondary view configuration types that application supports.

If there are any duplicated view configuration types in the array of `enabledViewConfigurationTypes`, the runtime **must** return error `XR_ERROR_VALIDATION_FAILURE`.

If there are any primary view configuration types in the array of `enabledViewConfigurationTypes`, the runtime **must** return error `XR_ERROR_VALIDATION_FAILURE`.

If there are any secondary view configuration types not returned by xrEnumerateViewConfigurations in the array of `enabledViewConfigurationTypes`, the runtime **must** return error `XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED`.

Valid Usage (Implicit)

- The `XR_MSFT_secondary_view_configuration` extension **must** be enabled prior to using XrSecondaryViewConfigurationSessionBeginInfoMSFT
- `type` **must** be `XR_TYPE_SECONDARY_VIEW_CONFIGURATION_SESSION_BEGIN_INFO_MSFT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `enabledViewConfigurationTypes` **must** be a pointer to an array of `viewConfigurationCount` valid XrViewConfigurationType values
- The `viewConfigurationCount` parameter **must** be greater than `0`

12.49.5. Per-frame active view configurations

The runtime then tells the application at each xrWaitFrame function call which of the enabled secondary view configurations are active for that frame. When extension structure XrSecondaryViewConfigurationFrameStateMSFT is chained to the XrFrameState::next pointer, the runtime writes into this structure the state of each enabled secondary view configuration.

The XrSecondaryViewConfigurationFrameStateMSFT structure returns whether the enabled view configurations are active or inactive.

It is defined as as:

```
typedef struct XrSecondaryViewConfigurationFrameStateMSFT {
    XrStructureType                            type;
    void*                                      next;
    uint32_t                                   viewConfigurationCount;
    XrSecondaryViewConfigurationStateMSFT*    viewConfigurationStates;
} XrSecondaryViewConfigurationFrameStateMSFT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `viewConfigurationCount` is the number of elements in `viewConfigurationStates`.
- `viewConfigurationStates` is an array of XrSecondaryViewConfigurationStateMSFT structures.

The array size `viewConfigurationCount` in the XrSecondaryViewConfigurationFrameStateMSFT structure **must** be the same as the array size enabled through XrSecondaryViewConfigurationSessionBeginInfoMSFT when calling xrBeginSession earlier, otherwise the runtime **must** return error `XR_ERROR_VALIDATION_FAILURE`.

Valid Usage (Implicit)

- The `XR_MSFT_secondary_view_configuration` extension **must** be enabled prior to using XrSecondaryViewConfigurationFrameStateMSFT
- `type` **must** be `XR_TYPE_SECONDARY_VIEW_CONFIGURATION_FRAME_STATE_MSFT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `viewConfigurationStates` **must** be a pointer to an array of `viewConfigurationCount` XrSecondaryViewConfigurationStateMSFT structures
- The `viewConfigurationCount` parameter **must** be greater than `0`

The XrSecondaryViewConfigurationStateMSFT structure returns the state of an enabled secondary view configurations.

```
typedef struct XrSecondaryViewConfigurationStateMSFT {
    XrStructureType            type;
    void*                      next;
    XrViewConfigurationType    viewConfigurationType;
    XrBool32                   active;
} XrSecondaryViewConfigurationStateMSFT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `viewConfigurationType` is a XrViewConfigurationType that represents the returned state.
- `active` is an `XrBool32` returns whether the secondary view configuration is active and displaying frames to users.

When a secondary view configuration becomes active, the application **should** render its secondary views as soon as possible, by getting their view transforms and FOV using xrLocateViews and then submitting composition layers to xrEndFrame through the XrSecondaryViewConfigurationFrameEndInfoMSFT extension structure. When a secondary view configuration changes from inactive to active, the runtime **may** change XrViewConfigurationView of the given view configuration such as the recommended image width or height. An application **should** query for latest XrViewConfigurationView through xrEnumerateViewConfigurationViews function for the secondary view configuration and consider recreating swapchain images if necessary. The runtime

must not change the XrViewConfigurationView, including recommended image width and height of a secondary view configuration when `active` remains true until the secondary view configuration deactivated or the session has ended.

If necessary, the application **can** take longer than a frame duration to prepare by calling xrEndFrame without submitting layers for that secondary view configuration until ready. The runtime **should** delay the underlying scenario managed by the secondary view configuration until the application begins submitting frames with layers for that configuration. The active secondary view configuration composed output is undefined if the application stops submitting frames with layers for a secondary view configuration while `active` remains true.

When the runtime intends to conclude a secondary view configuration, for example when user stops video capture, the runtime makes the view configuration inactive by setting the corresponding `active` in the XrSecondaryViewConfigurationStateMSFT structure to false.

Valid Usage (Implicit)

- The `XR_MSFT_secondary_view_configuration` extension **must** be enabled prior to using XrSecondaryViewConfigurationStateMSFT
- `type` **must** be `XR_TYPE_SECONDARY_VIEW_CONFIGURATION_STATE_MSFT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `viewConfigurationType` **must** be a valid XrViewConfigurationType value

12.49.6. Locate and inspect view states of secondary view configurations

When the application calls xrLocateViews, it **can** use XrViewLocateInfo::viewConfigurationType field to query the view locations and projections for any enabled XrViewConfigurationType for the running session.

The runtime **must** return `XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED` from xrLocateViews if the specified XrViewConfigurationType is not enabled for the running session using XrSecondaryViewConfigurationSessionBeginInfoMSFT when calling xrBeginSession.

If the view configuration is supported but not active, as indicated in XrSecondaryViewConfigurationFrameStateMSFT, xrLocateViews will successfully return, but the resulting `XrViewState` **may** have `XR_VIEW_STATE_ORIENTATION_TRACKED_BIT` and `XR_VIEW_STATE_ORIENTATION_TRACKED_BIT` unset.

12.49.7. Submit composition layers to secondary view configurations

The application **should** submit layers each frame for all active secondary view configurations using the xrEndFrame function, by chaining the XrSecondaryViewConfigurationFrameEndInfoMSFT structure to the next pointer of XrFrameEndInfo structure.

The XrSecondaryViewConfigurationFrameEndInfoMSFT structure is defined as as:

```
typedef struct XrSecondaryViewConfigurationFrameEndInfoMSFT {
    XrStructureType                                      type;
    const void*                                          next;
    uint32_t                                             viewConfigurationCount;
    const XrSecondaryViewConfigurationLayerInfoMSFT*     viewConfigurationLayersInfo;
} XrSecondaryViewConfigurationFrameEndInfoMSFT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `viewConfigurationCount` is the number of elements in `viewConfigurationLayersInfo`.
- `viewConfigurationLayersInfo` is an array of XrSecondaryViewConfigurationLayerInfoMSFT, containing composition layers to be submitted for the specified active view configuration.

The view configuration type in each XrSecondaryViewConfigurationLayerInfoMSFT must be one of the view configurations enabled when calling xrBeginSession in XrSecondaryViewConfigurationSessionBeginInfoMSFT, or else the runtime **must** return error `XR_ERROR_SECONDARY_VIEW_CONFIGURATION_TYPE_NOT_ENABLED_MSFT`.

The view configuration type in each XrSecondaryViewConfigurationLayerInfoMSFT must not be the primary view configuration in this session, or else the runtime **must** return error `XR_ERROR_LAYER_INVALID`. The primary view configuration layers continue to be submitted through XrFrameEndInfo directly.

If the view configuration is not active, as indicated in XrSecondaryViewConfigurationFrameStateMSFT, the composition layers submitted to this view configuration **may** be ignored by the runtime. Applications **should** avoid rendering into secondary views when the view configuration is inactive.

Valid Usage (Implicit)

- The `XR_MSFT_secondary_view_configuration` extension **must** be enabled prior to using XrSecondaryViewConfigurationFrameEndInfoMSFT
- `type` **must** be `XR_TYPE_SECONDARY_VIEW_CONFIGURATION_FRAME_END_INFO_MSFT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `viewConfigurationLayersInfo` **must** be a pointer to an array of `viewConfigurationCount` valid XrSecondaryViewConfigurationLayerInfoMSFT structures
- The `viewConfigurationCount` parameter **must** be greater than `0`

The application should submit an XrSecondaryViewConfigurationLayerInfoMSFT in XrSecondaryViewConfigurationFrameEndInfoMSFT for each active secondary view configuration type when calling xrEndFrame.

The XrSecondaryViewConfigurationLayerInfoMSFT structure is defined as as:

```
typedef struct XrSecondaryViewConfigurationLayerInfoMSFT {
    XrStructureType                               type;
    const void*                                   next;
    XrViewConfigurationType                       viewConfigurationType;
    XrEnvironmentBlendMode                        environmentBlendMode;
    uint32_t                                      layerCount;
    const XrCompositionLayerBaseHeader* const*    layers;
} XrSecondaryViewConfigurationLayerInfoMSFT;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- viewConfigurationType is XrViewConfigurationType to which the composition layers will be displayed.
- environmentBlendMode is the XrEnvironmentBlendMode value representing the desired environment blend mode for this view configuration.
- layerCount is the number of composition layers in this frame for the secondary view configuration type. The maximum supported layer count is identified by XrSystemGraphicsProperties::maxLayerCount. If layerCount is greater than the maximum supported layer count then XR_ERROR_LAYER_LIMIT_EXCEEDED is returned. The runtime **must** support at least XR_MIN_COMPOSITION_LAYERS_SUPPORTED layers.
- layers is a pointer to an array of XrCompositionLayerBaseHeader pointers.

This structure is similar to the XrFrameEndInfo structure, with an extra XrViewConfigurationType field to specify the view configuration for which the submitted layers will be rendered.

The application **should** render its content for both the primary and secondary view configurations using the same predictedDisplayTime reported by xrWaitFrame. The runtime **must** treat both the primary views and secondary views as being submitted for the same displayTime specified in the call to xrEndFrame.

For layers such as quad layers whose content is identical across view configurations, the application **can** submit the same XrCompositionLayerBaseHeader structures to multiple view configurations in the same xrEndFrame function call.

For each frame, the application **should** only render and submit layers for the secondary view configurations that were active that frame, as indicated in the XrSecondaryViewConfigurationFrameStateMSFT filled in for that frame's xrWaitFrame call. The runtime **must** ignore composition layers submitted for an inactive view configuration.

Valid Usage (Implicit)

- The `XR_MSFT_secondary_view_configuration` extension **must** be enabled prior to using XrSecondaryViewConfigurationLayerInfoMSFT
- `type` **must** be `XR_TYPE_SECONDARY_VIEW_CONFIGURATION_LAYER_INFO_MSFT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `viewConfigurationType` **must** be a valid XrViewConfigurationType value
- `environmentBlendMode` **must** be a valid XrEnvironmentBlendMode value
- `layers` **must** be a pointer to an array of `layerCount` valid XrCompositionLayerBaseHeader -based structures. See also: XrCompositionLayerCubeKHR, XrCompositionLayerCylinderKHR, XrCompositionLayerEquirect2KHR, XrCompositionLayerEquirectKHR, XrCompositionLayerProjection, XrCompositionLayerQuad
- The `layerCount` parameter **must** be greater than `0`

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_SECONDARY_VIEW_CONFIGURATION_SESSION_BEGIN_INFO_MSFT`
- `XR_TYPE_SECONDARY_VIEW_CONFIGURATION_STATE_MSFT`
- `XR_TYPE_SECONDARY_VIEW_CONFIGURATION_FRAME_STATE_MSFT`
- `XR_TYPE_SECONDARY_VIEW_CONFIGURATION_FRAME_END_INFO_MSFT`
- `XR_TYPE_SECONDARY_VIEW_CONFIGURATION_LAYER_INFO_MSFT`
- `XR_ERROR_SECONDARY_VIEW_CONFIGURATION_TYPE_NOT_ENABLED_MSFT`

New Enums

New Structures

- XrSecondaryViewConfigurationSessionBeginInfoMSFT
- XrSecondaryViewConfigurationStateMSFT
- XrSecondaryViewConfigurationFrameStateMSFT
- XrSecondaryViewConfigurationFrameEndInfoMSFT
- XrSecondaryViewConfigurationLayerInfoMSFT

New Functions

Issues

Version History

- Revision 1, 2019-07-30 (Yin Li)
  - Initial extension description

12.50. XR_MSFT_spatial_anchor

Name String

`XR_MSFT_spatial_anchor`

Extension Type

Instance extension

Registered Extension Number

40

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Overview

This extension allows an application to create a **spatial anchor**, an arbitrary freespace point in the user's physical environment that will then be tracked by the runtime. The runtime **should** then adjust the position and orientation of that anchor's origin over time as needed, independently of all other spaces and anchors, to ensure that it maintains its original mapping to the real world.

```
XR_DEFINE_HANDLE(XrSpatialAnchorMSFT)
```

Spatial anchors are often used in combination with an `UNBOUNDED_MSFT` reference space. `UNBOUNDED_MSFT` reference spaces adjust their origin as necessary to keep the viewer's coordinates relative to the space's origin stable. Such adjustments maintain the visual stability of content currently near the viewer, but may cause content placed far from the viewer to drift in its alignment to the real world by the time the user moves close again. By creating an `XrSpatialAnchorMSFT` where a piece of content is placed and then always rendering that content relative to its anchor's space, an application can ensure that each piece of content stays at a fixed location in the environment.

The xrCreateSpatialAnchorMSFT function is defined as:

```
XrResult xrCreateSpatialAnchorMSFT(
    XrSession                                   session,
    const XrSpatialAnchorCreateInfoMSFT*        createInfo,
    XrSpatialAnchorMSFT*                        anchor);
```

Parameter Descriptions

- session is a handle to an XrSession.
- createInfo is a pointer to an XrSpatialAnchorCreateInfoMSFT structure containing information about how to create the anchor.
- anchor is a pointer to a handle in which the created XrSpatialAnchorMSFT is returned.

Creates an XrSpatialAnchorMSFT handle representing a spatial anchor that will track a fixed location in the physical world over time. That real-world location is specified by the position and orientation of the specified pose within space at time.

If space cannot be located relative to the environment at the moment of the call to xrCreateSpatialAnchorMSFT, the runtime **must** return XR_ERROR_CREATE_SPATIAL_ANCHOR_FAILED_MSFT.

After the anchor is created, the runtime **should** then adjust its position and orientation over time relative to other spaces so as to maintain maximum alignment to its original real-world location, even if that changes the anchor's relationship to the original space used to initialize it.

Valid Usage (Implicit)

- The XR_MSFT_spatial_anchor extension **must** be enabled prior to calling xrCreateSpatialAnchorMSFT
- session **must** be a valid XrSession handle
- createInfo **must** be a pointer to a valid XrSpatialAnchorCreateInfoMSFT structure
- anchor **must** be a pointer to an XrSpatialAnchorMSFT handle

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_POSE_INVALID
- XR_ERROR_CREATE_SPATIAL_ANCHOR_FAILED_MSFT
- XR_ERROR_TIME_INVALID
- XR_ERROR_SESSION_LOST
- XR_SESSION_LOSS_PENDING

The XrSpatialAnchorCreateInfoMSFT structure is defined as:

```
typedef struct XrSpatialAnchorCreateInfoMSFT {
    XrStructureType    type;
    const void*        next;
    XrSpace            space;
    XrPosef            pose;
    XrTime             time;
} XrSpatialAnchorCreateInfoMSFT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `space` is a handle to the XrSpace in which `pose` is specified.
- `pose` is the XrPosef within `space` at `time` that specifies the point in the real world used to initialize the new anchor.
- `time` is the `XrTime` at which `pose` will be evaluated within `space`.

Valid Usage (Implicit)

- The `XR_MSFT_spatial_anchor` extension **must** be enabled prior to using XrSpatialAnchorCreateInfoMSFT
- `type` **must** be `XR_TYPE_SPATIAL_ANCHOR_CREATE_INFO_MSFT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `space` **must** be a valid XrSpace handle

The xrCreateSpatialAnchorSpaceMSFT function is defined as:

```
XrResult xrCreateSpatialAnchorSpaceMSFT(
    XrSession                                   session,
    const XrSpatialAnchorSpaceCreateInfoMSFT*   createInfo,
    XrSpace*                                    space);
```

Parameter Descriptions

- `session` is a handle to an XrSession.
- `createInfo` is a pointer to an XrSpatialAnchorSpaceCreateInfoMSFT structure containing information about how to create the anchor.
- `space` is a pointer to a handle in which the created XrSpace is returned.

Creates an XrSpace handle based on a spatial anchor. Application **can** provide an XrPosef to define the position and orientation of the new space's origin relative to the anchor's natural origin.

Multiple XrSpace handles may exist for a given XrSpatialAnchorMSFT simultaneously, up to some limit imposed by the runtime. The XrSpace handle must be eventually freed via the xrDestroySpace function or by destroying the parent XrSpatialAnchorMSFT handle.

Valid Usage (Implicit)

- The `XR_MSFT_spatial_anchor` extension **must** be enabled prior to calling xrCreateSpatialAnchorSpaceMSFT
- `session` **must** be a valid XrSession handle
- `createInfo` **must** be a pointer to a valid XrSpatialAnchorSpaceCreateInfoMSFT structure
- `space` **must** be a pointer to an XrSpace handle

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_OUT_OF_MEMORY`
- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_POSE_INVALID`
- `XR_ERROR_SESSION_LOST`
- `XR_SESSION_LOSS_PENDING`

The XrSpatialAnchorSpaceCreateInfoMSFT structure is defined as:

```
typedef struct XrSpatialAnchorSpaceCreateInfoMSFT {
    XrStructureType       type;
    const void*           next;
    XrSpatialAnchorMSFT   anchor;
    XrPosef               poseInAnchorSpace;
} XrSpatialAnchorSpaceCreateInfoMSFT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `anchor` is a handle to an XrSpatialAnchorMSFT previously created with xrCreateSpatialAnchorMSFT.
- `poseInAnchorSpace` is an XrPosef defining the position and orientation of the new space's origin relative to the anchor's natural origin.

Valid Usage (Implicit)

- The `XR_MSFT_spatial_anchor` extension **must** be enabled prior to using XrSpatialAnchorSpaceCreateInfoMSFT
- `type` **must** be `XR_TYPE_SPATIAL_ANCHOR_SPACE_CREATE_INFO_MSFT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `anchor` **must** be a valid XrSpatialAnchorMSFT handle

The xrDestroySpatialAnchorMSFT function is defined as:

```
XrResult xrDestroySpatialAnchorMSFT(
    XrSpatialAnchorMSFT                         anchor);
```

Parameter Descriptions

- `anchor` is a handle to an XrSpatialAnchorMSFT previously created by xrCreateSpatialAnchorMSFT.

XrSpatialAnchorMSFT handles are destroyed using xrDestroySpatialAnchorMSFT. By destroying an anchor, the runtime **can** stop spending resources used to maintain tracking for that anchor's origin.

Valid Usage (Implicit)

- The `XR_MSFT_spatial_anchor` extension **must** be enabled prior to calling xrDestroySpatialAnchorMSFT
- `anchor` **must** be a valid XrSpatialAnchorMSFT handle

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_FUNCTION_UNSUPPORTED`

New Object Types

XrSpatialAnchorMSFT

New Flag Types

New Enum Constants

XrObjectType enumeration is extended with:

- `XR_OBJECT_TYPE_SPATIAL_ANCHOR_MSFT`

XrStructureType enumeration is extended with:

- `XR_TYPE_SPATIAL_ANCHOR_CREATE_INFO_MSFT`

XrResult enumeration is extended with:

- `XR_ERROR_CREATE_SPATIAL_ANCHOR_FAILED_MSFT`

New Enums

New Structures

XrSpatialAnchorCreateInfoMSFT

XrSpatialAnchorSpaceCreateInfoMSFT

New Functions

xrCreateSpatialAnchorMSFT

xrCreateSpatialAnchorSpaceMSFT

xrDestroySpatialAnchorMSFT

Issues

Version History

- Revision 1, 2019-07-30 (Alex Turner)
  - Initial extension description

12.51. XR_MSFT_spatial_graph_bridge

Name String

`XR_MSFT_spatial_graph_bridge`

Extension Type

Instance extension

Registered Extension Number

50

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Contributors

Yin Li, Microsoft
Alex Turner, Microsoft
David Fields, Microsoft
Darryl Gough, Microsoft

Overview

This extension enables applications to create XrSpace handles from other Windows Mixed Reality device platform libraries or APIs. These libraries represent a spatially tracked point, also known as a "spatial graph node", with a GUID value.

The xrCreateSpatialGraphNodeSpaceMSFT function creates an XrSpace handle for a given spatial graph node type and ID.

```
XrResult xrCreateSpatialGraphNodeSpaceMSFT(
    XrSession                                   session,
    const XrSpatialGraphNodeSpaceCreateInfoMSFT* createInfo,
    XrSpace*                                    space);
```

Parameter Descriptions

- session is the XrSession which will use the created space.
- createInfo is an XrSpatialGraphNodeSpaceCreateInfoMSFT specifying the space to be created.
- space is the returned XrSpace handle for the given spatial node ID.

Valid Usage (Implicit)

- The XR_MSFT_spatial_graph_bridge extension **must** be enabled prior to calling xrCreateSpatialGraphNodeSpaceMSFT
- session **must** be a valid XrSession handle
- createInfo **must** be a pointer to a valid XrSpatialGraphNodeSpaceCreateInfoMSFT structure
- space **must** be a pointer to an XrSpace handle

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_SESSION_LOST
- XR_SESSION_LOSS_PENDING
- XR_ERROR_POSE_INVALID

The XrSpatialGraphNodeSpaceCreateInfoMSFT structure is used with xrCreateSpatialGraphNodeSpaceMSFT to create an XrSpace handle for a given spatial node type and node ID.

```
typedef struct XrSpatialGraphNodeSpaceCreateInfoMSFT {
    XrStructureType               type;
    const void*                   next;
    XrSpatialGraphNodeTypeMSFT    nodeType;
    uint8_t                       nodeId[16];
    XrPosef                       pose;
} XrSpatialGraphNodeSpaceCreateInfoMSFT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `nodeType` is an XrSpatialGraphNodeTypeMSFT specifying the spatial node type.
- `nodeId` is a global unique identifier (a.k.a. GUID or 16 byte array), representing the spatial node that is being tracked.
- `pose` is an XrPosef defining the position and orientation of the new space's origin within the natural reference frame of the spatial graph node.

Valid Usage (Implicit)

- The `XR_MSFT_spatial_graph_bridge` extension **must** be enabled prior to using XrSpatialGraphNodeSpaceCreateInfoMSFT
- `type` **must** be `XR_TYPE_SPATIAL_GRAPH_NODE_SPACE_CREATE_INFO_MSFT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `nodeType` **must** be a valid XrSpatialGraphNodeTypeMSFT value

The enum XrSpatialGraphNodeTypeMSFT describes the types of spatial graph nodes.

```
typedef enum XrSpatialGraphNodeTypeMSFT {
    XR_SPATIAL_GRAPH_NODE_TYPE_STATIC_MSFT = 1,
    XR_SPATIAL_GRAPH_NODE_TYPE_DYNAMIC_MSFT = 2,
    XR_SPATIAL_GRAPH_NODE_TYPE_MAX_ENUM_MSFT = 0x7FFFFFFF
} XrSpatialGraphNodeTypeMSFT;
```

There are two types of spatial graph nodes: static and dynamic.

Static spatial nodes track the pose of a fixed location in the world relative to reference spaces. The tracking of static nodes **may** slowly adjust the pose over time for better accuracy but the pose is relatively stable in the short term, such as between rendering frames. For example, a QR code tracking library can use a static node to represent the location of the tracked QR code. Static spatial nodes are represented by `XR_SPATIAL_GRAPH_NODE_TYPE_STATIC_MSFT`.

Dynamic spatial nodes track the pose of a physical object that moves continuously relative to reference spaces. The pose of dynamic spatial nodes **can** be very different within the duration of a rendering frame. It is important for the application to use the correct timestamp to query the space location using xrLocateSpace. For example, a color camera mounted in front of a HMD is also tracked by the HMD so a web camera library can use a dynamic node to represent the camera location. Dynamic spatial nodes are represented by `XR_SPATIAL_GRAPH_NODE_TYPE_DYNAMIC_MSFT`.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_SPATIAL_GRAPH_NODE_SPACE_CREATE_INFO_MSFT`

New Enums

XrSpatialGraphNodeTypeMSFT

New Structures

XrSpatialGraphNodeSpaceCreateInfoMSFT

New Functions

xrCreateSpatialGraphNodeSpaceMSFT

Issues

Version History

- Revision 1, 2019-10-31 (Yin LI)
  - Initial extension description

12.52. XR_MSFT_unbounded_reference_space

Name String

`XR_MSFT_unbounded_reference_space`

Extension Type

Instance extension

Registered Extension Number

39

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Overview

This extension allows an application to create an `UNBOUNDED_MSFT` reference space. This reference space enables the viewer to move freely through a complex environment, often many meters from where they started, while always optimizing for coordinate system stability near the viewer. This is done by allowing the origin of the reference space to drift as necessary to keep the viewer's coordinates relative to the space's origin stable.

To create an `UNBOUNDED_MSFT` reference space, the application **can** pass `XR_REFERENCE_SPACE_TYPE_UNBOUNDED_MSFT` to xrCreateReferenceSpace.

The `UNBOUNDED_MSFT` reference space establishes a world-locked origin, gravity-aligned to exclude pitch and roll, with +Y up, +X to the right, and -Z forward. This space begins with an arbitrary initial position and orientation, which the runtime **may** define to be either the initial position at app launch or some other initial zero position. Unlike a `STAGE` reference space, the runtime **may** place the origin of an `UNBOUNDED_MSFT` reference space at any height, rather than fixing it at the floor. This is because the viewer may move through various rooms and levels of their environment, each of which has a different floor height. Runtimes **should** not automatically adjust the position of the origin when the viewer moves to a room with a different floor height.

`UNBOUNDED_MSFT` space is useful when an app needs to render **world-scale** content that spans beyond the bounds of a single `STAGE`, for example, an entire floor or multiple floors of a building.

An `UNBOUNDED_MSFT` space maintains stability near the viewer by slightly adjusting its origin over time. The runtime **must** not queue the XrEventDataReferenceSpaceChangePending event in response to these minor adjustments.

When views, controllers or other spaces experience tracking loss relative to the UNBOUNDED_MSFT space, runtimes **should** continue to provide inferred or last-known position and orientation values. These inferred poses can, for example, be based on neck model updates, inertial dead reckoning, or a last-known position, so long as it is still reasonable for the application to use that pose. While a runtime is providing position data, it **must** continue to set XR_SPACE_LOCATION_POSITION_VALID_BIT and XR_VIEW_STATE_POSITION_VALID_BIT but it **can** clear XR_SPACE_LOCATION_POSITION_TRACKED_BIT and XR_VIEW_STATE_POSITION_TRACKED_BIT to indicate that the position is inferred or last-known in this way.

When tracking is recovered, runtimes **should** snap the pose of other spaces back into position relative to the UNBOUNDED_MSFT space's original origin. However, if tracking recovers into a new tracking volume in which the original origin can no longer be located (e.g. the viewer moved through a dark hallway and regained tracking in a new room), the runtime **may** recenter the origin arbitrarily, for example moving the origin to coincide with the viewer. If such recentering occurs, the runtime **must** queue the XrEventDataReferenceSpaceChangePending event with poseValid set to false.

If the viewer moves far enough away from the origin of an UNBOUNDED_MSFT reference space that floating point error would introduce noticeable error when locating the viewer within that space, the runtime **may** recenter the space's origin to a new location closer to the viewer. If such recentering occurs, the runtime **must** queue the XrEventDataReferenceSpaceChangePending event with poseValid set to true.

Runtimes **must** support the UNBOUNDED_MSFT reference space when this extension is enabled.

New Object Types

New Flag Types

New Enum Constants

XrReferenceSpaceType enumeration is extended with:

- XR_REFERENCE_SPACE_TYPE_UNBOUNDED_MSFT

New Enums

New Structures

New Functions

Issues

Version History

- Revision 1, 2019-07-30 (Alex Turner)
  - Initial extension description

12.53. XR_OCULUS_android_session_state_enable

Name String

`XR_OCULUS_android_session_state_enable`

Extension Type

Instance extension

Registered Extension Number

45

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Overview

This extension enables the integration of the Android session lifecycle and an OpenXR runtime session state. Some OpenXR runtimes may require this extension to transition the application to the session READY or STOPPING state.

Applications that run on an Android system with this extension enabled have a different OpenXR Session state flow.

On Android, it is the Android Activity lifecycle that will dictate when the system is ready for the application to begin or end its session, not the runtime.

When XR_OCULUS_android_session_state is enabled, the following changes are made to Session State handling:

- The runtime does not determine when the application's session should be moved to the ready state, `XR_SESSION_STATE_READY`. The application should not wait to receive the `XR_SESSION_STATE_READY` session state changed event before beginning a session. Instead, the application should begin their session once there is a surface and the activity is resumed.
- The application should not call xrRequestExitSession to request the session move to the stopping state, `XR_SESSION_STATE_STOPPING`. xrRequestExitSession will return `XR_ERROR_VALIDATION_FAILURE` if called.
- The application should not wait to receive the `XR_SESSION_STATE_STOPPING` session state changed event before ending a session. Instead, the application should end its session once the surface is destroyed or the activity is paused.
- The runtime will not transition to `XR_SESSION_STATE_READY` or `XR_SESSION_STATE_STOPPING` as the state is implicit from the Android activity and surface lifecycles.

Android Activity life cycle

An Android Activity can only be in the session running state while the activity is in the resumed state. The following shows how beginning and ending an XR session fits into the Android Activity life cycle.

```
    1.  VrActivity::onCreate() <---------+
    2.  VrActivity::onStart() <-------+  |
    3.  VrActivity::onResume() <---+  |  |
    4.  xrBeginSession()           |  |  |
    5.  xrEndSession()             |  |  |
    6.  VrActivity::onPause() -----+  |  |
    7.  VrActivity::onStop() ---------+  |
    8.  VrActivity::onDestroy() ---------+
```

Android Surface life cycle

An Android Activity can only be in the session running state while there is a valid Android Surface. The following shows how beginning and ending an XR session fits into the Android Surface life cycle.

```
    1.  VrActivity::surfaceCreated() <----+
    2.  VrActivity::surfaceChanged()      |
    3.  xrBeginSession()                  |
    4.  xrEndSession()                    |
    5.  VrActivity::surfaceDestroyed() ---+
```

Note that the life cycle of a surface is not necessarily tightly coupled with the life cycle of an activity. These two life cycles may interleave in complex ways. Usually surfaceCreated() is called after onResume() and surfaceDestroyed() is called between onPause() and onDestroy(). However, this is not guaranteed and, for instance, surfaceDestroyed() may be called after onDestroy() or even before onPause().

An Android Activity is only in the resumed state with a valid Android Surface between surfaceChanged() or onResume(), whichever comes last, and surfaceDestroyed() or onPause(), whichever comes first. In other words, a XR application will typically begin the session from surfaceChanged() or onResume(), whichever comes last, and end the session from surfaceDestroyed() or onPause(), whichever comes first.

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

New Functions

Issues

Version History

- Revision 1, 2019-08-16 (Cass Everitt)
  - Initial extension description

12.54. XR_VALVE_analog_threshold

Name String

`XR_VALVE_analog_threshold`

Extension Type

Instance extension

Registered Extension Number

80

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2020-06-29

IP Status

No known IP claims.

Contributors

Joe Ludwig, Valve

Overview

This extension allows the application to control the threshold and haptic feedback applied to an analog to digital conversion. See XrInteractionProfileAnalogThresholdVALVE for more information.

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

The XrInteractionProfileAnalogThresholdVALVE structure is an input struct which can be added to the next chain of XrInteractionProfileSuggestedBinding to the thresholds and haptic feedback to use for a suggested binding.

```
typedef struct XrInteractionProfileAnalogThresholdVALVE {
    XrStructureType              type;
    const void*                  next;
    XrAction                     action;
    XrPath                       binding;
    float                        onThreshold;
    float                        offThreshold;
    const XrHapticBaseHeader*    onHaptic;
    const XrHapticBaseHeader*    offHaptic;
} XrInteractionProfileAnalogThresholdVALVE;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `action` is the handle of an action in the suggested binding list.
- `binding` is the input path used for the specified action in the suggested binding list.
- `onThreshold` is the value between 0.0 and 1.0 at which the runtime **must** consider the binding to be true. The binding must remain true until the input analog value falls below `offThreshold`.
- `offThreshold` is the value between 0.0 and 1.0 at which the runtime **must** consider the binding to be false if it was previous true.
- `onHaptic` is the haptic output that the runtime **must** trigger when the binding changes from false to true. If this field is NULL, the runtime **must** not trigger any haptic output on the threshold. This field **can** point to any supported sub-type of XrHapticBaseHeader.
- `offHaptic` is the haptic output that the runtime **must** trigger when the binding changes from true to false. If this field is NULL, the runtime **must** not trigger any haptic output on the threshold. This field **can** point to any supported sub-type of XrHapticBaseHeader.

Applications can chain an XrInteractionProfileAnalogThresholdVALVE struct on the next chain of any xrSuggestInteractionProfileBindings call for each analog to boolean conversion for which it wants to set the threshold. If a threshold struct is present for a given conversion, the runtime **must** use those thresholds instead of applying its own whenever it is using the binding suggested by the application.

`onThreshold` and `offThreshold` permit allow the application to specify that it wants hysteresis to be applied to the threshold operation. If `onThreshold` is smaller than `offThreshold`, the runtime **must** return `XR_ERROR_VALIDATION_FAILURE`.

`onHaptic` and `offHaptic` allow the application to specify that it wants automatic haptic feedback to be generated when the boolean output of the threshold operation changes from false to true or vice versa. If these fields are not NULL, the runtime **must** trigger a haptic output with the specified characteristics. If the device has multiple haptic outputs, the runtime **should** use the haptic output that is most appropriate for the specified input path.

If a suggested binding with `action` and `binding` is not in the binding list for this interaction profile, the runtime **must** return `XR_ERROR_PATH_UNSUPPORTED`.

Valid Usage (Implicit)

- The `XR_VALVE_analog_threshold` extension **must** be enabled prior to using XrInteractionProfileAnalogThresholdVALVE
- `type` **must** be `XR_TYPE_INTERACTION_PROFILE_ANALOG_THRESHOLD_VALVE`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `action` **must** be a valid XrAction handle
- If `onHaptic` is not `NULL`, `onHaptic` **must** be a pointer to a valid XrHapticBaseHeader-based structure. See also: XrHapticVibration
- If `offHaptic` is not `NULL`, `offHaptic` **must** be a pointer to a valid XrHapticBaseHeader-based structure. See also: XrHapticVibration

New Functions

Issues

Version History

- Revision 1, 2020-06-29 (Joe Ludwig)
  - Initial version.

12.55. XR_VARJO_composition_layer_depth_test

Name String

`XR_VARJO_composition_layer_depth_test`

Extension Type

Instance extension

Registered Extension Number

123

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0
- Requires `XR_KHR_composition_layer_depth`

Last Modified Date

2021-02-16

IP Status

No known IP claims.

Contributors

Sergiy Dubovik, Varjo Technologies
Antti Hirvonen, Varjo Technologies
Rémi Arnaud, Varjo Technologies

Overview

This extension enables depth-based layer composition inside the compositor.

Core OpenXR specifies that layer compositing must happen in the layer submission order (as described in Compositing). However, an application may want to composite the final image against the other layers based on depth information for proper occlusion. Layers can now provide depth information that will be used to calculate occlusion between those layers, as well as with the environment depth estimator (`XR_VARJO_environment_depth_estimation`) when enabled.

This extension defines a new type, XrCompositionLayerDepthTestVARJO, which can be chained to XrCompositionLayerProjection in order to activate this functionality. An application must also specify a range where depth testing will happen, potentially covering only a subset of the full depth range.

Composition

Layer composition rules change when this extension is enabled.

If the application does not chain XrCompositionLayerDepthTestVARJO, "painter's algorithm" such as described in Compositing must be used for layer composition.

Overall, composition should be performed in the following way:

1. Layers must be composited in the submission order. The compositor must track the depth value nearest to the virtual camera. Initial value for the nearest depth should be infinity.
2. If the currently processed layer does not contain depth, compositor should composite the layer against the previous layers with "painter’s algorithm" and move to the next layer.
3. If the layer depth or the active nearest depth fall inside the depth test range of the layer, the compositor must perform depth test against the layer and active depth. If the layer depth is less or equal than the active depth, layer is composited normally with the previous layers and active depth is updated to match the layer depth. Otherwise the layer pixel is discarded, and compositor should move to composite the next layer.

Example

Mixed reality applications may want to show hands on top of the rendered VR content. For this purpose the application should enable environment depth estimation (see `XR_VARJO_environment_depth_estimation` extension) and depth testing with range 0m to 1m.

The following code illustrates how to enable depth testing:

```
XrCompositionLayerProjection layer{XR_TYPE_COMPOSITION_LAYER_PROJECTION};
layer.space = ...;
layer.viewCount = ...;
layer.views = ...;
layer.layerFlags = ...;

XrCompositionLayerDepthTestVARJO depthTest{XR_TYPE_COMPOSITION_LAYER_DEPTH_TEST_VARJO,
layer.next};
depthTest.depthTestRangeNearZ = 0.0f; // in meters
depthTest.depthTestRangeFarZ = 1.0f; // in meters
layer.next = &depthTest
```

New Structures

Applications **can** enable depth testing by adding XrCompositionLayerDepthTestVARJO to the `next` chain for all XrCompositionLayerProjectionView structures in the given layer in addition to XrCompositionLayerDepthInfoKHR. Missing XrCompositionLayerDepthInfoKHR automatically disables the depth testing functionality.

The XrCompositionLayerDepthTestVARJO structure is defined as:

```
typedef struct XrCompositionLayerDepthTestVARJO {
    XrStructureType    type;
    const void*        next;
    float              depthTestRangeNearZ;
    float              depthTestRangeFarZ;
} XrCompositionLayerDepthTestVARJO;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `depthTestRangeNearZ` in a non-negative distance in meters that specifies the lower bound of the range where depth testing should be performed. Must be less than `depthTestRangeFarZ`. Value of zero means that there is no lower bound.
- `depthTestRangeFarZ` is a positive distance in meters that specifies the upper bound of the range where depth testing should be performed. Must be greater than `depthTestRangeNearZ`. Value of floating point positive infinity means that there is no upper bound.

Valid Usage (Implicit)

- The `XR_VARJO_composition_layer_depth_test` extension **must** be enabled prior to using XrCompositionLayerDepthTestVARJO
- `type` **must** be `XR_TYPE_COMPOSITION_LAYER_DEPTH_TEST_VARJO`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_COMPOSITION_LAYER_DEPTH_TEST_VARJO`

Version History

- Revision 1, 2021-02-16 (Sergiy Dubovik)
  - Initial extension description

12.56. XR_VARJO_environment_depth_estimation

Name String

`XR_VARJO_environment_depth_estimation`

Extension Type

Instance extension

Registered Extension Number

124

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2021-02-17

IP Status

No known IP claims.

Contributors

Sergiy Dubovik, Varjo Technologies
Antti Hirvonen, Varjo Technologies
Rémi Arnaud, Varjo Technologies

Overview

This extension provides a mechanism for enabling depth estimation of the environment in the runtime-supplied compositor. This is an extension to `XR_ENVIRONMENT_BLEND_MODE_ALPHA_BLEND` mode to not only use the color but also depth for composition of the final image.

Mixed reality applications might want to mix real and virtual content based on the depth information for proper occlusion. XR hardware and runtime may offer various ways to estimate the depth of the environment inside the compositor. When this estimation is enabled, the compositor can generate properly occluded final image when layers are submitted with depth information (both `XR_KHR_composition_layer_depth` and `XR_VARJO_composition_layer_depth_test`).

This extension defines a new function, xrSetEnvironmentDepthEstimationVARJO, which **can** be used to toggle environment depth estimation in the compositor. Toggling depth estimation is an asynchronous operation and the feature **may** not be activated immediately. Function can be called immediately after the session is created. Composition of the environment layer follows the rules as described in `XR_VARJO_composition_layer_depth_test`.

New Structures

The xrSetEnvironmentDepthEstimationVARJO function is defined as:

```
XrResult xrSetEnvironmentDepthEstimationVARJO(
    XrSession                                   session,
    XrBool32                                    enabled);
```

Parameter Descriptions

- session is an XrSession handle previously created with xrCreateSession.
- enabled is a boolean that specifies whether depth estimation functionality should be activated. Compositor will disable depth estimation functionality if environment blend mode is not XR_ENVIRONMENT_BLEND_MODE_ALPHA_BLEND and will enable the functionality when environment blend mode is set to XR_ENVIRONMENT_BLEND_MODE_ALPHA_BLEND.

Valid Usage (Implicit)

- The XR_VARJO_environment_depth_estimation extension **must** be enabled prior to calling xrSetEnvironmentDepthEstimationVARJO
- session **must** be a valid XrSession handle

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_FEATURE_UNSUPPORTED
- XR_ERROR_FUNCTION_UNSUPPORTED

New Functions

Version History

- Revision 1, 2021-02-16 (Sergiy Dubovik)
  - Initial extension description

12.57. XR_VARJO_foveated_rendering

Name String

`XR_VARJO_foveated_rendering`

Extension Type

Instance extension

Registered Extension Number

122

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0
- Requires `XR_VARJO_quad_views`

Last Modified Date

2020-12-16

IP Status

No known IP claims.

Contributors

Sergiy Dubovik, Varjo Technologies
Rémi Arnaud, Varjo Technologies
Antti Hirvonen, Varjo Technologies

12.57.1. Overview

Varjo headsets provide extremely high pixel density displays in the center area of the display, blended with a high density display covering the rest of the field of view. If the application has to provide a single image per eye, that would cover the entire field of view, at the highest density it would be extremely resource intensive, and in fact impossible for the most powerful desktop GPUs to render in real time. So instead Varjo introduced the `XR_VARJO_quad_views` extension enabling the application to provide two separate images for the two screen areas, resulting in a significant reduction in processing, for pixels that could not even been seen.

This extension goes a step further by enabling the application to only generate the density that can be seen by the user, which is another big reduction compared to the density that can be displayed, using dedicated eye tracking.

This extension requires XR_VARJO_quad_views extension to be enabled.

An application using this extension to enable foveated rendering will take the following steps to prepare:

1. Enable XR_VARJO_quad_views and XR_VARJO_foveated_rendering extensions.
2. Query system properties in order to determine if system supports foveated rendering.
3. Query texture sizes for foveated rendering.

In the render loop, for each frame, an application using this extension **should**

1. Check if rendering gaze is available using xrLocateSpace.
2. Enable foveated rendering when xrLocateViews is called.

12.57.2. Inspect system capability

An application **can** inspect whether the system is capable of foveated rendering by chaining a XrSystemFoveatedRenderingPropertiesVARJO structure to the XrSystemProperties structure when calling xrGetSystemProperties.

```
typedef struct XrSystemFoveatedRenderingPropertiesVARJO {
    XrStructureType    type;
    void*              next;
    XrBool32           supportsFoveatedRendering;
} XrSystemFoveatedRenderingPropertiesVARJO;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `supportsFoveatedRendering` is an `XrBool32`, indicating if current system is capable of performoning foveated rendering.

The runtime **should** return `XR_TRUE` for `supportsFoveatedRendering` when rendering gaze is available in the system. An application **should** avoid using foveated rendering functionality when supportsFoveatedRendering is XR_FALSE.

Valid Usage (Implicit)

- The XR_VARJO_foveated_rendering extension **must** be enabled prior to using XrSystemFoveatedRenderingPropertiesVARJO
- type **must** be XR_TYPE_SYSTEM_FOVEATED_RENDERING_PROPERTIES_VARJO
- next **must** be NULL or a valid pointer to the next structure in a structure chain

12.57.3. Determine foveated texture sizes

Foveated textures **may** have different sizes and aspect ratio compared to non-foveated textures. In order to determine recommended foveated texture size, an application **can** chain XrFoveatedViewConfigurationViewVARJO to XrViewConfigurationView and set foveatedRenderingActive to XR_TRUE. Since an application using foveated rendering with this extension has to render 4 views, XR_VARJO_quad_views **must** be enabled along with this extension when XrInstance is created.

First and second views are non foveated views (covering whole field of view of HMD), third (left eye) and fourth (right eye) are foveated e.g. following gaze.

```
typedef struct XrFoveatedViewConfigurationViewVARJO {
    XrStructureType    type;
    void*              next;
    XrBool32           foveatedRenderingActive;
} XrFoveatedViewConfigurationViewVARJO;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- foveatedRenderingActive is an XrBool32, indicating if the runtime should return foveated view configuration view.

Valid Usage (Implicit)

- The XR_VARJO_foveated_rendering extension **must** be enabled prior to using XrFoveatedViewConfigurationViewVARJO
- type **must** be XR_TYPE_FOVEATED_VIEW_CONFIGURATION_VIEW_VARJO
- next **must** be NULL or a valid pointer to the next structure in a structure chain

For example:

```
XrViewConfigurationType viewConfigType... // Select
XR_VIEW_CONFIGURATION_TYPE_PRIMARY_QUAD_VARJO
uint32_t viewCount;
CHK_XR(xrEnumerateViewConfigurationViews(instance, systemId, viewConfigType, 0,
&viewCount, nullptr));
// Non-foveated rendering views dimensions
std::vector<XrViewConfigurationView> configViews(viewCount,
{XR_TYPE_VIEW_CONFIGURATION_VIEW});
CHK_XR(xrEnumerateViewConfigurationViews(instance, systemId, viewConfigType, viewCount,
&viewCount, configViews.data()));

// Foveated rendering views dimensions
std::vector<XrViewConfigurationView> foveatedViews;
if (foveatedRenderingProperties.supportsFoveatedRendering && viewConfigType ==
XR_VIEW_CONFIGURATION_TYPE_PRIMARY_QUAD_VARJO) {
  std::vector<XrFoveatedViewConfigurationViewVARJO> requestFoveatedConfig{4,
{XR_TYPE_FOVEATED_VIEW_CONFIGURATION_VIEW_VARJO, nullptr, XR_TRUE}};
  foveatedViews = std::vector<XrViewConfigurationView>{4,
{XR_TYPE_VIEW_CONFIGURATION_VIEW}};
  for (size_t i = 0; i < 4; i++) {
    foveatedViews[i].next = &requestFoveatedConfig[i];
  }
  CHK_XR(xrEnumerateViewConfigurationViews(instance, systemId, viewConfigType, viewCount,
&viewCount, foveatedViews.data()));
}
```

Example 2. Note

Applications using this extension are encouraged to create 2 sets of swapchains or one big enough set of swapchains and 2 sets of viewports. One set will be used when rendering gaze is not available and other one will be used when foveated rendering and rendering gaze is available. Using foveated textures **may** not provide optimal visual quality when rendering gaze is not available.

12.57.4. Rendering gaze status

Extension defines new reference space type - XR_REFERENCE_SPACE_TYPE_COMBINED_EYE_VARJO which **should** be used to determine whether rendering gaze is available. After calling xrLocateSpace, application **should** inspect XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT bit. If it's set, rendering gaze is available otherwise not.

```
...
// Create needed spaces
XrSpace renderGazeSpace;
XrSpace viewSpace;
XrReferenceSpaceCreateInfo createViewSpaceInfo{XR_TYPE_REFERENCE_SPACE_CREATE_INFO};
createViewSpaceInfo.referenceSpaceType = XR_REFERENCE_SPACE_TYPE_VIEW;
createViewSpaceInfo.poseInReferenceSpace.orientation.w = 1.0f;
CHK_XR(xrCreateReferenceSpace(session, &createViewSpaceInfo, &viewSpace));

XrReferenceSpaceCreateInfo createReferenceSpaceInfo{XR_TYPE_REFERENCE_SPACE_CREATE_INFO};
createReferenceSpaceInfo.referenceSpaceType = XR_REFERENCE_SPACE_TYPE_COMBINED_EYE_VARJO;
createReferenceSpaceInfo.poseInReferenceSpace.orientation.w = 1.0f;
CHK_XR(xrCreateReferenceSpace(session, &createReferenceSpaceInfo, &renderGazeSpace));


...
// Query rendering gaze status
XrSpaceLocation renderGazeLocation{XR_TYPE_SPACE_LOCATION};
CHK_XR(xrLocateSpace(renderGazeSpace, viewSpace, frameState.predictedDisplayTime,
&renderGazeLocation));

const bool foveationActive = (renderGazeLocation.locationFlags &
XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT) != 0;

if(foveationActive){
  // Rendering gaze is available
} else {
  // Rendering gaze is not available
}
```

12.57.5. Request foveated field of view

For each frame, the application indicates if the runtime will return foveated or non-foveated field of view. This is done by chaining XrViewLocateFoveatedRenderingVARJO to XrViewLocateInfo.

```
typedef struct XrViewLocateFoveatedRenderingVARJO {
    XrStructureType    type;
    const void*        next;
    XrBool32           foveatedRenderingActive;
} XrViewLocateFoveatedRenderingVARJO;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `foveatedRenderingActive` is an `XrBool32`, indicating if runtime should return foveated FoV.

The runtime **must** return foveated field of view when `foveatedRenderingActive` is `XR_TRUE`.

Valid Usage (Implicit)

- The `XR_VARJO_foveated_rendering` extension **must** be enabled prior to using XrViewLocateFoveatedRenderingVARJO
- `type` **must** be `XR_TYPE_VIEW_LOCATE_FOVEATED_RENDERING_VARJO`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

```
XrViewState viewState{XR_TYPE_VIEW_STATE};
uint32_t viewCapacityInput = static_cast<uint32_t>(views.size());
uint32_t viewCountOutput;
XrViewLocateInfo viewLocateInfo{XR_TYPE_VIEW_LOCATE_INFO};
viewLocateInfo.displayTime = frameState.predictedDisplayTime;
viewLocateInfo.space = appSpace;
XrViewLocateFoveatedRenderingVARJO
viewLocateFoveatedRendering{XR_TYPE_VIEW_LOCATE_FOVEATED_RENDERING_VARJO};
viewLocateFoveatedRendering.foveatedRenderingActive = foveationActive;
viewLocateInfo.next = &viewLocateFoveatedRendering;

CHK_XR(xrLocateViews(session, &viewLocateInfo, &viewState, viewCapacityInput,
&viewCountOutput, views.data()));
```

New Structures

- XrViewLocateFoveatedRenderingVARJO
- XrFoveatedViewConfigurationViewVARJO

* XrSystemFoveatedRenderingPropertiesVARJO

New Enum Constants

XrStructureType enumeration is extended with:

* XR_TYPE_VIEW_LOCATE_FOVEATED_RENDERING_VARJO
* XR_TYPE_FOVEATED_VIEW_CONFIGURATION_VIEW_VARJO
* XR_TYPE_SYSTEM_FOVEATED_RENDERING_PROPERTIES_VARJO

XrReferenceSpaceType enumeration is extended with:

* XR_REFERENCE_SPACE_TYPE_COMBINED_EYE_VARJO

Version History

* Revision 1, 2020-12-16 (Sergiy Dubovik)
  * Initial extension description

12.58. XR_VARJO_quad_views

Name String

XR_VARJO_quad_views

Extension Type

Instance extension

Registered Extension Number

38

Revision

1

Extension and Version Dependencies

* Requires OpenXR 1.0

Last Modified Date

2019-04-16

IP Status

No known IP claims.

Contributors

Sergiy Dubovik, Varjo Technologies
Rémi Arnaud, Varjo Technologies

Robert Menzel, NVIDIA

12.58.1. Overview

This extension adds a new view configuration type - XR_VIEW_CONFIGURATION_TYPE_PRIMARY_QUAD_VARJO to XrViewConfigurationType which can be returned by xrEnumerateViewConfigurations to indicate that the runtime supports 4 viewports.

In this configuration each eye consists of two viewports of which one is smaller (in terms of field of view) of the other and fully included inside of the larger FoV one. The small FoV viewport however can have a higher resolution with respect to the same field of view in the outer viewport. The motivation is special hardware which superimposes a smaller, high resolution screen for the fovea region onto a larger screen for the periphery.

The runtime guarantees that the inner viewport of each eye is fully inside of the outer viewport.

To enumerate the 4 views xrEnumerateViewConfigurationViews can be used. The first two views (XrViewConfigurationView) will be for the left and right eyes for the outer viewport. The views 2 and 3 are for the left and right eyes for the inner viewport.

The relative position of the inner views relative to the outer views can change at run-time.

The runtime might blend between the views at the edges, so the application should not omit the inner field of view from being generated in the outer view.

New Object Types

New Flag Types

New Enum Constants

XrViewConfigurationType enumeration is extended with:

- XR_VIEW_CONFIGURATION_TYPE_PRIMARY_QUAD_VARJO

New Enums

New Structures

New Functions

Issues

Version History

- Revision 1, 2019-04-16 (Sergiy Dubovik)
  - Initial draft

Chapter 13. List of Provisional Extensions

- XR_EXTX_overlay
- XR_MNDX_egl_enable <<<

13.1. XR_EXTX_overlay

Name String

XR_EXTX_overlay

Extension Type

Instance extension

Registered Extension Number

34

Revision

5

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2021-01-13

IP Status

No known IP claims.

Contributors

Mark Young, LunarG
Jules Blok, Epic
Jared Cheshier, Pluto VR
Nick Whiting, Epic
Brad Grantham, LunarG

Overview

Application developers may desire to implement an OpenXR application that renders content on top of another OpenXR application. These additional applications will execute in a separate process, create a separate session, generate separate content, but want the OpenXR runtime to composite their content on top of the main OpenXR application. Examples of these applications might include:

- A debug environment outputting additional content
- A Store application that hovers to one side of the user's view

- A interactive HUD designed to expose additional chat features

This extension introduces the concept of "Overlay Sessions" in order to expose this usage model.

This extension allows:

- An application to identify when the current sessions composition layers will be applied during composition
- The ability for an overlay session to get information about what is going on with the main application

To enable the functionality of this extension, an application **must** pass the name of the extension into xrCreateInstance via the XrInstanceCreateInfo `enabledExtensionNames` parameter as indicated in the Extensions section.

To create an overlay session, an application **must** pass an XrSessionCreateInfoOverlayEXTX structure to xrCreateSession via the XrSessionCreateInfo structure's `next` parameter.

An overlay application should not assume that the values returned to it by xrWaitFrame in `predictedDisplayTime` in XrFrameState will be the same as the values returned to the main application or even correlated.

13.1.1. Overlay Session Layer Placement

Since one or more sessions may be active at the same time, this extension provides the ability for the application to identify when the frames of the current session will be composited into the final frame.

The XrSessionCreateInfoOverlayEXTX `sessionLayersPlacement` parameter provides information on when the sessions composition layers should be applied to the final composition frame. The larger the value passed into `sessionLayersPlacement`, the closer to the front this session's composition layers will appear (relative to other overlay session's composition layers). The smaller the value of `sessionLayersPlacement`, the further to the back this session's composition's layers will appear. The main session's composition layers will always be composited first, resulting in any overlay content being composited on top of the main application's content.

If `sessionLayersPlacement` is 0, then the runtime will always attempt to composite that session's composition layers first. If `sessionLayersPlacement` is UINT32_MAX, then the runtime will always attempt to composite that session's composition layers last. If two or more overlay sessions are created with the same `sessionLayersPlacement` value, then the newer session's will be treated as if they had a slightly higher value of `sessionLayersPlacement` than the previous sessions with the same value. This should result in the newest overlay session being composited closer to the user than the older session.

The following image hopefully will provide any further clarification you need:

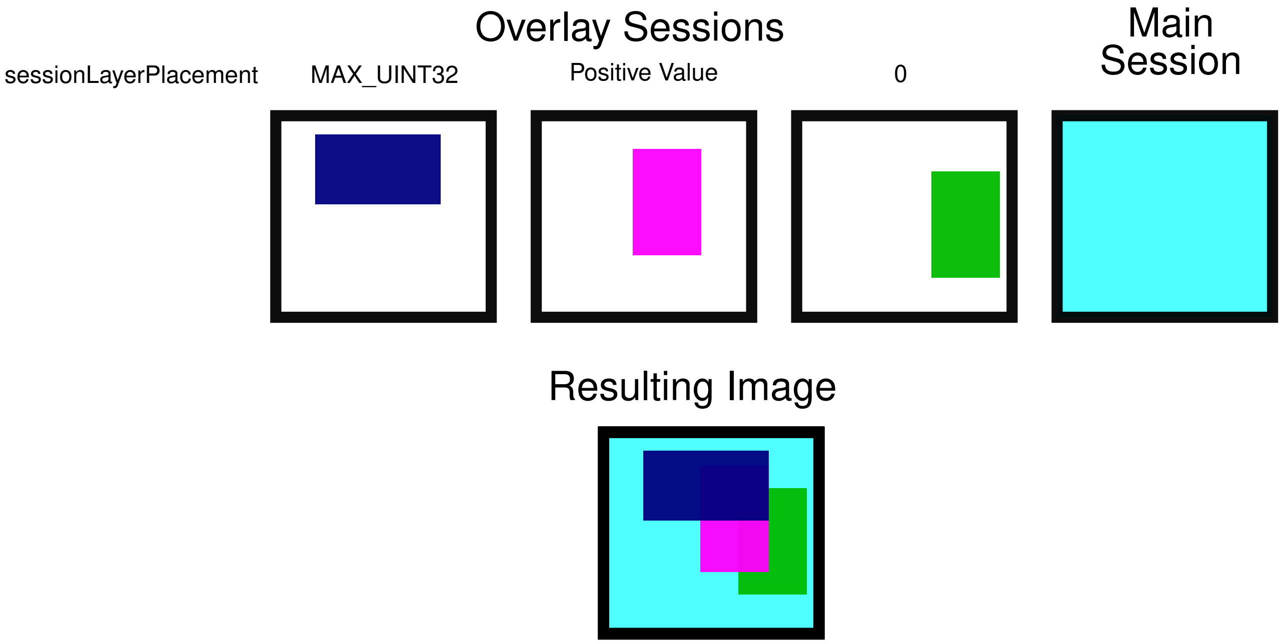


Figure 4. Overlay Composition Order

13.1.2. Main Session Behavior Event

Since an overlay session's intends to work in harmony with a main session, some information needs to be provided from that main session to the overlay session.

The XrEventDataMainSessionVisibilityChangedEXTX event structure provides information on the visibility of the main session as well as some additional flags which can be used to adjust overlay behavior.

If XR_KHR_composition_layer_depth is enabled in the main session, then XrEventDataMainSessionVisibilityChangedEXTX `flags` **should** contain the value: `XR_OVERLAY_MAIN_SESSION_ENABLED_COMPOSITION_LAYER_INFO_DEPTH_BIT_EXTX`. If the overlay session also enables XR_KHR_composition_layer_depth, then when both sessions are visible, the runtime can integrate their projection layer content together using depth information as described in the extension. However, if either the main session or the overlay do not enable the extension, then composition behavior will continue as if neither one enabled the extension.

13.1.3. Modifications to the OpenXR Specification

When this extension is enabled, certain core behaviors defined in the OpenXR specification must change as defined below:

Modifications to Composition

The Compositing section description of the composition process will be changed if this extension is enabled. If this extension is enabled, and there is only one active session, then there is no change. However, if this extension is enabled, and there are multiple active sessions, then the composition will occur in order based on the overlay session's XrSessionCreateInfoOverlayEXTX::sessionLayersPlacement value as described in the table below:

Table 6. Overlay Session Composition Order

Session Type	XrSessionCreateInfoOverlayEXTX::sessionLayersPlacement	Composited
Overlay Session	UINT32_MAX	*Composited last, appears in front of all other XrSessions*
Overlay Session	<Positive value>	
Overlay Session	0	
Non-overlay Session	N/A	*Composited first, appears behind all other XrSessions*

The above change only applies to when a session's composition layers are applied to the resulting image. The order in which composition layers are handled internal to a session does not change. However, once the sessions have been properly ordered, the runtime should behave as if all the composition layers have been placed into a single list (maintaining the separation of viewport images) and treat them as if they were from one original session. From this point forward, the composition behavior of the resulting composition layers is the same whether or not this extension is enabled.

If the overlay session is created as part of an XrInstance which has enabled the XR_KHR_composition_layer_depth extension, and a XrCompositionLayerDepthInfoKHR structure has been provided to one or more composition layers, then it intends for those layers to be composited into the final image using that depth information. This composition occurs as defined in the XR_KHR_composition_layer_depth extension. However, this is only possible if the main session has provided depth buffer information as part of its swapchain. In the event that a main session does not provide depth buffer information as part of its swapchain, then overlay application's composition layers containing depth information will be composited as if they did not contain that information.

Modifications to xrEndFrame Behavior

Frame Submission currently states that if xrEndFrame is called with no layers, then the runtime should clear the VR display.

If this extension is enabled, the above statement is now only true if the session is not an overlay session. If the session is an overlay session, and it provides 0 layers in the call to xrEndFrame, then the runtime will just ignore the overlay session for the current frame.

Modifications to Input Synchronization

If a runtime supports this extension, it **must** separate input tracking on a per-session basis. This means that reading the input from one active session does not disturb the input information that can be read by another active session. This may require duplicating events to more than one session.

New Object Types

None

New Flag Types

```
typedef XrFlags64 XrOverlayMainSessionFlagsEXTX;
```

```
// Flag bits for XrOverlayMainSessionFlagsEXTX
static const XrOverlayMainSessionFlagsEXTX
XR_OVERLAY_MAIN_SESSION_ENABLED_COMPOSITION_LAYER_INFO_DEPTH_BIT_EXTX = 0x00000001;
```

```
typedef XrFlags64 XrOverlaySessionCreateFlagsEXTX;
```

```
// Flag bits for XrOverlaySessionCreateFlagsEXTX
```

New Enum Constants

XrStructureType enumeration is extended with:

- XR_TYPE_SESSION_CREATE_INFO_OVERLAY_EXTX
- XR_TYPE_EVENT_DATA_MAIN_SESSION_VISIBILITY_CHANGED_EXTX

New Enums

- XR_OVERLAY_MAIN_SESSION_ENABLED_COMPOSITION_LAYER_INFO_DEPTH_BIT_EXTX

New Structures

```
typedef struct XrSessionCreateInfoOverlayEXTX {
    XrStructureType                 type;
    const void*                     next;
    XrOverlaySessionCreateFlagsEXTX    createFlags;
    uint32_t                        sessionLayersPlacement;
} XrSessionCreateInfoOverlayEXTX;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `createFlags` is `0` or one or more XrOverlaySessionCreateFlagBitsEXTX which indicate various characteristics desired for the overlay session.
- `sessionLayersPlacement` is a value indicating the desired placement of the session's composition layers in terms of other sessions.

Valid Usage (Implicit)

- The `XR_EXTX_overlay` extension **must** be enabled prior to using XrSessionCreateInfoOverlayEXTX
- `type` **must** be `XR_TYPE_SESSION_CREATE_INFO_OVERLAY_EXTX`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `createFlags` **must** be `0`

```
typedef struct XrEventDataMainSessionVisibilityChangedEXTX {
    XrStructureType                 type;
    const void*                     next;
    XrBool32                        visible;
    XrOverlayMainSessionFlagsEXTX    flags;
} XrEventDataMainSessionVisibilityChangedEXTX;
```

Receiving the XrEventDataMainSessionVisibilityChangedEXTX event structure indicates that the main session has gained or lost visibility. This can occur in many cases, one typical example is when a user switches from one OpenXR application to another. See XrEventDataMainSessionVisibilityChangedEXTX for more information on the standard behavior. This structure contains additional information on the main session including `flags` which indicate additional state information of the main session.

Currently, the only flag value supplied is XR_OVERLAY_MAIN_SESSION_ENABLED_COMPOSITION_LAYER_INFO_DEPTH_BIT_EXTX which indicates if the main session has enabled the XR_KHR_composition_layer_depth extension.

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- visible is an XrBool32 which indicates if session is now visible or is not.
- flags is 0 or one or more XrOverlayMainSessionFlagBitsEXTX which indicates various state information for the main session.

Valid Usage (Implicit)

- The XR_EXTX_overlay extension **must** be enabled prior to using XrEventDataMainSessionVisibilityChangedEXTX
- type **must** be XR_TYPE_EVENT_DATA_MAIN_SESSION_VISIBILITY_CHANGED_EXTX
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- flags **must** be a valid combination of XrOverlayMainSessionFlagBitsEXTX values
- flags **must** not be 0

New Functions

None

New Function Pointers

None

Issues

None

Version History

- Revision 1, 2018-11-05 (Mark Young)
  - Initial draft
- Revision 2, 2020-02-12 (Brad Grantham)
  - Name change, remove overlay bool, add flags

- Revision 3, 2020-03-05 (Brad Grantham)
  - Name change
- Revision 4, 2020-03-23 (Brad Grantham)
  - Fix enums
- Revision 5, 2021-01-13 (Brad Grantham)
  - Remove bit requesting synchronized display times

13.2. XR_MNDX_egl_enable

Name String

`XR_MNDX_egl_enable`

Extension Type

Instance extension

Registered Extension Number

49

Revision

1

Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2020-05-21

IP Status

No known IP claims.

Contributors

Jakob Bornecrantz, Collabora
Drew DeVault, Individual
Simon Ser, Individual

Overview

This extension must be provided by runtimes supporting applications using the EGL API to create rendering contexts.

- XR_USE_PLATFORM_EGL

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_GRAPHICS_BINDING_EGL_MNDX`

New Enums

New Structures

The XrGraphicsBindingEGLMNDX structure is defined as:

```
typedef struct XrGraphicsBindingEGLMNDX {
    XrStructureType              type;
    const void*                  next;
    PFNEGLGETPROCADDRESSPROC     getProcAddress;
    EGLDisplay                   display;
    EGLConfig                    config;
    EGLContext                   context;
} XrGraphicsBindingEGLMNDX;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `getProcAddress` is a valid function pointer to `eglGetProcAddress`.
- `display` is a valid EGL `EGLDisplay`.
- `config` is a valid EGL `EGLConfig`.
- `context` is a valid EGL `EGLContext`.

When creating an EGL based XrSession, the application will provide a pointer to an XrGraphicsBindingEGLMNDX structure in the `next` chain of the XrSessionCreateInfo.

The required window system configuration define to expose this structure type is XR_USE_PLATFORM_EGL.

Valid Usage (Implicit)

- The XR_MNDX_egl_enable extension **must** be enabled prior to using XrGraphicsBindingEGLMNDX
- type **must** be XR_TYPE_GRAPHICS_BINDING_EGL_MNDX
- next **must** be NULL or a valid pointer to the next structure in a structure chain
- getProcAddress **must** be a valid PFNEGLGETPROCADDRESSPROC value
- display **must** be a valid EGLDisplay value
- config **must** be a valid EGLConfig value
- context **must** be a valid EGLContext value

New Functions

Issues

Version History

- Revision 1, 2020-05-20 (Jakob Bornecrantz)
  - Initial draft

Appendix

Code Style Conventions

These are the code style conventions used in this specification to define the API.

Conventions

- Enumerants and defines are all upper case with words separated by an underscore.
- Neither type, function or member names contain underscores.
- Structure members start with a lower case character and each consecutive word starts with a capital.
- A structure that has a pointer to an array includes a structure member named `fooCount` of type `uint32_t` to denote the number of elements in the array of `foo`.
- A structure that has a pointer to an array lists the `fooCount` member first and then the array pointer.
- Unless a negative value has a clearly defined meaning all `fooCount` variables are unsigned.
- Function parameters that are modified are always listed last.

Prefixes are used in the API to denote specific semantic meaning of names, or as a label to avoid name clashes, and are explained here:

Prefix	Description
`XR_`	Enumerants and defines are prefixed with these characters.
`Xr`	Non-function-pointer types are prefixed with these characters.
`xr`	Functions are prefixed with these characters.
`PFN_xr`	Function pointer types are prefixed with these characters.

Application Binary Interface

This section describes additional definitions and conventions that define the application binary interface.

Structure Types

```
typedef enum XrStructureType {
    XR_TYPE_UNKNOWN = 0,
    XR_TYPE_API_LAYER_PROPERTIES = 1,
    XR_TYPE_EXTENSION_PROPERTIES = 2,
    XR_TYPE_INSTANCE_CREATE_INFO = 3,
    XR_TYPE_SYSTEM_GET_INFO = 4,
    XR_TYPE_SYSTEM_PROPERTIES = 5,
    XR_TYPE_VIEW_LOCATE_INFO = 6,
    XR_TYPE_VIEW = 7,
    XR_TYPE_SESSION_CREATE_INFO = 8,
    XR_TYPE_SWAPCHAIN_CREATE_INFO = 9,
    XR_TYPE_SESSION_BEGIN_INFO = 10,
    XR_TYPE_VIEW_STATE = 11,
    XR_TYPE_FRAME_END_INFO = 12,
    XR_TYPE_HAPTIC_VIBRATION = 13,
    XR_TYPE_EVENT_DATA_BUFFER = 16,
    XR_TYPE_EVENT_DATA_INSTANCE_LOSS_PENDING = 17,
    XR_TYPE_EVENT_DATA_SESSION_STATE_CHANGED = 18,
    XR_TYPE_ACTION_STATE_BOOLEAN = 23,
    XR_TYPE_ACTION_STATE_FLOAT = 24,
    XR_TYPE_ACTION_STATE_VECTOR2F = 25,
    XR_TYPE_ACTION_STATE_POSE = 27,
    XR_TYPE_ACTION_SET_CREATE_INFO = 28,
    XR_TYPE_ACTION_CREATE_INFO = 29,
    XR_TYPE_INSTANCE_PROPERTIES = 32,
    XR_TYPE_FRAME_WAIT_INFO = 33,
    XR_TYPE_COMPOSITION_LAYER_PROJECTION = 35,
    XR_TYPE_COMPOSITION_LAYER_QUAD = 36,
    XR_TYPE_REFERENCE_SPACE_CREATE_INFO = 37,
    XR_TYPE_ACTION_SPACE_CREATE_INFO = 38,
    XR_TYPE_EVENT_DATA_REFERENCE_SPACE_CHANGE_PENDING = 40,
    XR_TYPE_VIEW_CONFIGURATION_VIEW = 41,
    XR_TYPE_SPACE_LOCATION = 42,
    XR_TYPE_SPACE_VELOCITY = 43,
    XR_TYPE_FRAME_STATE = 44,
    XR_TYPE_VIEW_CONFIGURATION_PROPERTIES = 45,
    XR_TYPE_FRAME_BEGIN_INFO = 46,
    XR_TYPE_COMPOSITION_LAYER_PROJECTION_VIEW = 48,
    XR_TYPE_EVENT_DATA_EVENTS_LOST = 49,
    XR_TYPE_INTERACTION_PROFILE_SUGGESTED_BINDING = 51,
    XR_TYPE_EVENT_DATA_INTERACTION_PROFILE_CHANGED = 52,
    XR_TYPE_INTERACTION_PROFILE_STATE = 53,
    XR_TYPE_SWAPCHAIN_IMAGE_ACQUIRE_INFO = 55,
```

```
    XR_TYPE_SWAPCHAIN_IMAGE_WAIT_INFO = 56,
    XR_TYPE_SWAPCHAIN_IMAGE_RELEASE_INFO = 57,
    XR_TYPE_ACTION_STATE_GET_INFO = 58,
    XR_TYPE_HAPTIC_ACTION_INFO = 59,
    XR_TYPE_SESSION_ACTION_SETS_ATTACH_INFO = 60,
    XR_TYPE_ACTIONS_SYNC_INFO = 61,
    XR_TYPE_BOUND_SOURCES_FOR_ACTION_ENUMERATE_INFO = 62,
    XR_TYPE_INPUT_SOURCE_LOCALIZED_NAME_GET_INFO = 63,
    XR_TYPE_COMPOSITION_LAYER_CUBE_KHR = 1000006000,
    XR_TYPE_INSTANCE_CREATE_INFO_ANDROID_KHR = 1000008000,
    XR_TYPE_COMPOSITION_LAYER_DEPTH_INFO_KHR = 1000010000,
    XR_TYPE_VULKAN_SWAPCHAIN_FORMAT_LIST_CREATE_INFO_KHR = 1000014000,
    XR_TYPE_EVENT_DATA_PERF_SETTINGS_EXT = 1000015000,
    XR_TYPE_COMPOSITION_LAYER_CYLINDER_KHR = 1000017000,
    XR_TYPE_COMPOSITION_LAYER_EQUIRECT_KHR = 1000018000,
    XR_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT = 1000019000,
    XR_TYPE_DEBUG_UTILS_MESSENGER_CALLBACK_DATA_EXT = 1000019001,
    XR_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT = 1000019002,
    XR_TYPE_DEBUG_UTILS_LABEL_EXT = 1000019003,
    XR_TYPE_GRAPHICS_BINDING_OPENGL_WIN32_KHR = 1000023000,
    XR_TYPE_GRAPHICS_BINDING_OPENGL_XLIB_KHR = 1000023001,
    XR_TYPE_GRAPHICS_BINDING_OPENGL_XCB_KHR = 1000023002,
    XR_TYPE_GRAPHICS_BINDING_OPENGL_WAYLAND_KHR = 1000023003,
    XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_KHR = 1000023004,
    XR_TYPE_GRAPHICS_REQUIREMENTS_OPENGL_KHR = 1000023005,
    XR_TYPE_GRAPHICS_BINDING_OPENGL_ES_ANDROID_KHR = 1000024001,
    XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_ES_KHR = 1000024002,
    XR_TYPE_GRAPHICS_REQUIREMENTS_OPENGL_ES_KHR = 1000024003,
    XR_TYPE_GRAPHICS_BINDING_VULKAN_KHR = 1000025000,
    XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR = 1000025001,
    XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN_KHR = 1000025002,
    XR_TYPE_GRAPHICS_BINDING_D3D11_KHR = 1000027000,
    XR_TYPE_SWAPCHAIN_IMAGE_D3D11_KHR = 1000027001,
    XR_TYPE_GRAPHICS_REQUIREMENTS_D3D11_KHR = 1000027002,
    XR_TYPE_GRAPHICS_BINDING_D3D12_KHR = 1000028000,
    XR_TYPE_SWAPCHAIN_IMAGE_D3D12_KHR = 1000028001,
    XR_TYPE_GRAPHICS_REQUIREMENTS_D3D12_KHR = 1000028002,
    XR_TYPE_SYSTEM_EYE_GAZE_INTERACTION_PROPERTIES_EXT = 1000030000,
    XR_TYPE_EYE_GAZE_SAMPLE_TIME_EXT = 1000030001,
    XR_TYPE_VISIBILITY_MASK_KHR = 1000031000,
    XR_TYPE_EVENT_DATA_VISIBILITY_MASK_CHANGED_KHR = 1000031001,
    XR_TYPE_SESSION_CREATE_INFO_OVERLAY_EXTX = 1000033000,
    XR_TYPE_EVENT_DATA_MAIN_SESSION_VISIBILITY_CHANGED_EXTX = 1000033003,
    XR_TYPE_COMPOSITION_LAYER_COLOR_SCALE_BIAS_KHR = 1000034000,
    XR_TYPE_SPATIAL_ANCHOR_CREATE_INFO_MSFT = 1000039000,
    XR_TYPE_SPATIAL_ANCHOR_SPACE_CREATE_INFO_MSFT = 1000039001,
    XR_TYPE_VIEW_CONFIGURATION_DEPTH_RANGE_EXT = 1000046000,
    XR_TYPE_GRAPHICS_BINDING_EGL_MNDX = 1000048004,
```

```
    XR_TYPE_SPATIAL_GRAPH_NODE_SPACE_CREATE_INFO_MSFT = 1000049000,
    XR_TYPE_SYSTEM_HAND_TRACKING_PROPERTIES_EXT = 1000051000,
    XR_TYPE_HAND_TRACKER_CREATE_INFO_EXT = 1000051001,
    XR_TYPE_HAND_JOINTS_LOCATE_INFO_EXT = 1000051002,
    XR_TYPE_HAND_JOINT_LOCATIONS_EXT = 1000051003,
    XR_TYPE_HAND_JOINT_VELOCITIES_EXT = 1000051004,
    XR_TYPE_SYSTEM_HAND_TRACKING_MESH_PROPERTIES_MSFT = 1000052000,
    XR_TYPE_HAND_MESH_SPACE_CREATE_INFO_MSFT = 1000052001,
    XR_TYPE_HAND_MESH_UPDATE_INFO_MSFT = 1000052002,
    XR_TYPE_HAND_MESH_MSFT = 1000052003,
    XR_TYPE_HAND_POSE_TYPE_INFO_MSFT = 1000052004,
    XR_TYPE_SECONDARY_VIEW_CONFIGURATION_SESSION_BEGIN_INFO_MSFT = 1000053000,
    XR_TYPE_SECONDARY_VIEW_CONFIGURATION_STATE_MSFT = 1000053001,
    XR_TYPE_SECONDARY_VIEW_CONFIGURATION_FRAME_STATE_MSFT = 1000053002,
    XR_TYPE_SECONDARY_VIEW_CONFIGURATION_FRAME_END_INFO_MSFT = 1000053003,
    XR_TYPE_SECONDARY_VIEW_CONFIGURATION_LAYER_INFO_MSFT = 1000053004,
    XR_TYPE_SECONDARY_VIEW_CONFIGURATION_SWAPCHAIN_CREATE_INFO_MSFT = 1000053005,
    XR_TYPE_CONTROLLER_MODEL_KEY_STATE_MSFT = 1000055000,
    XR_TYPE_CONTROLLER_MODEL_NODE_PROPERTIES_MSFT = 1000055001,
    XR_TYPE_CONTROLLER_MODEL_PROPERTIES_MSFT = 1000055002,
    XR_TYPE_CONTROLLER_MODEL_NODE_STATE_MSFT = 1000055003,
    XR_TYPE_CONTROLLER_MODEL_STATE_MSFT = 1000055004,
    XR_TYPE_VIEW_CONFIGURATION_VIEW_FOV_EPIC = 1000059000,
    XR_TYPE_HOLOGRAPHIC_WINDOW_ATTACHMENT_MSFT = 1000063000,
    XR_TYPE_ANDROID_SURFACE_SWAPCHAIN_CREATE_INFO_FB = 1000070000,
    XR_TYPE_SWAPCHAIN_STATE_ANDROID_SURFACE_DIMENSIONS_FB = 1000071000,
    XR_TYPE_SWAPCHAIN_STATE_SAMPLER_OPENGL_ES_FB = 1000071001,
    XR_TYPE_INTERACTION_PROFILE_ANALOG_THRESHOLD_VALVE = 1000079000,
    XR_TYPE_HAND_JOINTS_MOTION_RANGE_INFO_EXT = 1000080000,
    XR_TYPE_LOADER_INIT_INFO_ANDROID_KHR = 1000089000,
    XR_TYPE_VULKAN_INSTANCE_CREATE_INFO_KHR = 1000090000,
    XR_TYPE_VULKAN_DEVICE_CREATE_INFO_KHR = 1000090001,
    XR_TYPE_VULKAN_GRAPHICS_DEVICE_GET_INFO_KHR = 1000090003,
    XR_TYPE_COMPOSITION_LAYER_EQUIRECT2_KHR = 1000091000,
    XR_TYPE_EVENT_DATA_DISPLAY_REFRESH_RATE_CHANGED_FB = 1000101000,
    XR_TYPE_SYSTEM_COLOR_SPACE_PROPERTIES_FB = 1000108000,
    XR_TYPE_BINDING_MODIFICATIONS_KHR = 1000120000,
    XR_TYPE_VIEW_LOCATE_FOVEATED_RENDERING_VARJO = 1000121000,
    XR_TYPE_FOVEATED_VIEW_CONFIGURATION_VIEW_VARJO = 1000121001,
    XR_TYPE_SYSTEM_FOVEATED_RENDERING_PROPERTIES_VARJO = 1000121002,
    XR_TYPE_COMPOSITION_LAYER_DEPTH_TEST_VARJO = 1000122000,
    XR_TYPE_GRAPHICS_BINDING_VULKAN2_KHR = XR_TYPE_GRAPHICS_BINDING_VULKAN_KHR,
    XR_TYPE_SWAPCHAIN_IMAGE_VULKAN2_KHR = XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR,
    XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN2_KHR = XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN_KHR,
    XR_STRUCTURE_TYPE_MAX_ENUM = 0x7FFFFFFF
} XrStructureType;
```

Most structures containing `type` members have a value of `type` matching the type of the structure, as described more fully in Valid Usage for Structure Types.

Note that all extension enums begin at the extension enum base of 1^{10} (base 10). Each extension is assigned a block of 1000 enums, starting at the enum base and arranged by the extension's index.

For example, if an extension with index 5 wants to use an enum value of 3, the final enum is computed by:

enum = enum_base + (enum_index - 1) * 1000 + enum_value 1000004003 = 1000000000 + 4 * 1000 + 3

Flag Types

Flag types are all bitmasks aliasing the base type `XrFlags64` and with corresponding bit flag types defining the valid bits for that flag, as described in Valid Usage for Flags. Flag types supported by the API include:

```
typedef XrFlags64 XrCompositionLayerFlags;
```

```
typedef XrFlags64 XrInputSourceLocalizedNameFlags;
```

```
typedef XrFlags64 XrInstanceCreateFlags;
```

```
typedef XrFlags64 XrSessionCreateFlags;
```

```
typedef XrFlags64 XrSpaceLocationFlags;
```

```
typedef XrFlags64 XrSpaceVelocityFlags;
```

```
typedef XrFlags64 XrSwapchainCreateFlags;
```

```
typedef XrFlags64 XrSwapchainUsageFlags;
```

```
typedef XrFlags64 XrViewStateFlags;
```

General Macro Definitions

This API is defined in C and uses "C" linkage. The `openxr.h` header file is opened with:

```
#ifdef __cplusplus
extern "C" {
#endif
```

and closed with:

```
#ifdef __cplusplus
}
#endif
```

The supplied `openxr.h` header defines a small number of C preprocessor macros that are described below.

Version Number Macros

Two version numbers are defined in `openxr.h`. Each is packed into a 32-bit integer as described in API Version Number Function-like Macros.

```
// OpenXR current version number.
#define XR_CURRENT_API_VERSION XR_MAKE_VERSION(1, 0, 16)
```

XR_CURRENT_API_VERSION is the current version of the OpenXR API.

API Version Number Function-like Macros

API Version Numbers are three components, packed into a single 64-bit integer. The following macros manipulate version components and packed version numbers.

```
#define XR_MAKE_VERSION(major, minor, patch) \
    ((((major) & 0xffffULL) << 48) | (((minor) & 0xffffULL) << 32) | ((patch) &
0xffffffffULL))
```

Parameter Descriptions

- `major` is the major version number, packed into the most-significant 16 bits.
- `minor` is the minor version number, packed into the second-most-significant group of 16 bits.
- `patch` is the patch version number, in the least-significant 32 bits.

XR_MAKE_VERSION constructs a packed 64-bit integer API version number from three components. The format used is described in API Version Numbers and Semantics.

This macro **can** be used when constructing the XrApplicationInfo::`apiVersion` parameter passed to xrCreateInstance.

```
#define XR_VERSION_MAJOR(version) (uint16_t)(((uint64_t)(version) >> 48)& 0xffffULL)
```

Parameter Descriptions

- `version` is a packed version number, such as those produced with XR_MAKE_VERSION.

XR_VERSION_MAJOR extracts the API major version number from a packed version number.

```
#define XR_VERSION_MINOR(version) (uint16_t)(((uint64_t)(version) >> 32) & 0xffffULL)
```

Parameter Descriptions

- `version` is a packed version number, such as those produced with XR_MAKE_VERSION.

XR_VERSION_MINOR extracts the API minor version number from a packed version number.

```
#define XR_VERSION_PATCH(version) (uint32_t)((uint64_t)(version) & 0xffffffffULL)
```

Parameter Descriptions

- `version` is a packed version number, such as those produced with XR_MAKE_VERSION.

XR_VERSION_PATCH extracts the API patch version number from a packed version number.

Handle and Atom Macros

```
#if !defined(XR_DEFINE_HANDLE)
#if (XR_PTR_SIZE == 8)
    #define XR_DEFINE_HANDLE(object) typedef struct object##_T* object;
#else
    #define XR_DEFINE_HANDLE(object) typedef uint64_t object;
#endif
#endif
```

Parameter Descriptions

- object is the name of the resulting C type.

XR_DEFINE_HANDLE defines a handle type, which is an opaque 64 bit value, which **may** be implemented as an opaque, distinct pointer type on platforms with 64 bit pointers.

For further details, see Handles.

```
#if !defined(XR_NULL_HANDLE)
#if (XR_PTR_SIZE == 8) && XR_CPP_NULLPTR_SUPPORTED
    #define XR_NULL_HANDLE nullptr
#else
    #define XR_NULL_HANDLE 0
#endif
#endif
```

XR_NULL_HANDLE is a reserved value representing a non-valid object handle. It **may** be passed to and returned from API functions only when specifically allowed.

```
#if !defined(XR_DEFINE_ATOM)
    #define XR_DEFINE_ATOM(object) typedef uint64_t object;
#endif
```

Parameter Descriptions

- object is the name of the resulting C type.

XR_DEFINE_ATOM defines an atom type, which is an opaque 64 bit integer.

Platform-Specific Macro Definitions

Additional platform-specific macros and interfaces are defined using the included `openxr_platform.h` file. These macros are used to control platform-dependent behavior, and their exact definitions are under the control of specific platform implementations of the API.

Platform-Specific Calling Conventions

On many platforms the following macros are empty strings, causing platform- and compiler-specific default calling conventions to be used.

XRAPI_ATTR is a macro placed before the return type of an API function declaration. This macro controls calling conventions for C++11 and GCC/Clang-style compilers.

XRAPI_CALL is a macro placed after the return type of an API function declaration. This macro controls calling conventions for MSVC-style compilers.

XRAPI_PTR is a macro placed between the (and * in API function pointer declarations. This macro also controls calling conventions, and typically has the same definition as XRAPI_ATTR or XRAPI_CALL, depending on the compiler.

Examples:

Function declaration:

```
XRAPI_ATTR <return_type> XRAPI_CALL <function_name>(<function_parameters>);
```

Function pointer type declaration:

```
typedef <return_type> (XRAPI_PTR *PFN_<function_name>(<function_parameters>);
```

Platform-Specific Header Control

If the XR_NO_STDINT_H macro is defined by the application at compile time, before including any OpenXR header, extended integer types normally found in `<stdint.h>` and used by the OpenXR headers, such as `uint8_t`, **must** also be defined (as `typedef` or with the preprocessor) before including any OpenXR header. Otherwise, `openxr.h` and related headers will not compile. If XR_NO_STDINT_H is not defined, the system-provided `<stdint.h>` is used to define these types. There is a fallback path for Microsoft Visual Studio version 2008 and earlier versions (which lack this header) that is automatically activated as needed.

Graphics API Header Control

Compile Time Symbol	Graphics API Name
`XR_USE_GRAPHICS_API_OPENGL`	OpenGL
`XR_USE_GRAPHICS_API_OPENGL_ES`	OpenGL ES
`XR_USE_GRAPHICS_API_VULKAN`	Vulkan
`XR_USE_GRAPHICS_API_D3D11`	Direct3D 11
`XR_USE_GRAPHICS_API_D3D12`	Direct3D 12

Window System Header Control

Compile Time Symbol	Window System Name
`XR_USE_PLATFORM_WIN32`	Microsoft Windows
`XR_USE_PLATFORM_XLIB`	X Window System Xlib
`XR_USE_PLATFORM_XCB`	X Window System Xcb
`XR_USE_PLATFORM_WAYLAND`	Wayland
`XR_USE_PLATFORM_ANDROID`	Android Native

Glossary

The terms defined in this section are used throughout this Specification. Capitalization is not significant for these definitions.

Term	Description
Application	The XR application which calls the OpenXR API to communicate with an OpenXR runtime.
Deprecated	A feature/extension is deprecated if it is no longer recommended as the correct or best way to achieve its intended purpose. Generally a newer feature/extension will have been created that solves the same problem - in cases where no newer alternative feature exists, justification should be provided.
Handle	An opaque integer or pointer value used to refer to an object. Each object type has a unique handle type.
Haptic	Haptic or kinesthetic communication recreates the sense of touch by applying forces, vibrations, or motions to the user.
In-Process	Something that executes in the application's process.
Instance	The top-level object, which represents the application's connection to the runtime. Represented by an XrInstance object.
Normalized	A value that is interpreted as being in the range [0,1], or a vector whose norm is in that range, as a result of being implicitly divided or scaled by some other value.
Out-Of-Process	Something that executes outside the application's process.

Term	Description
Promoted	A feature is promoted if it is taken from an older extension and made available as part of a new core version of the API, or a newer extension that is considered to be either as widely supported or more so. A promoted feature may have minor differences from the original such as: • It may be renamed • A small number of non-intrusive parameters may have been added • The feature may be advertised differently by device features • The author ID suffixes will be changed or removed as appropriate
Provisional	A feature is released provisionally in order to get wider feedback on the functionality before it is finalized. Provisional features may change in ways that break backwards compatibility, and thus are not recommended for use in production applications.
Required Extensions	Extensions that must be enabled alongside extensions dependent on them, or that must be enabled to use given hardware.
Runtime	The software which implements the OpenXR API and allows applications to interact with XR hardware.
Swapchain	A resource that represents a chain of images in device memory. Represented by an XrSwapchain object.
Swapchain Image	Each element in a swapchain. Commonly these are simple formatted 2D images, but in other cases they may be array images. Represented by a structure related to XrSwapchainImageBaseHeader.

Abbreviations

Abbreviations and acronyms are sometimes used in the API where they are considered clear and commonplace, and are defined here:

Abbreviation	Description
API	Application Programming Interface
AR	Augmented Reality
ER	Eye Relief
IAD	Inter Axial Distance
IPD	Inter Pupillary Distance
MR	Mixed Reality
OS	Operating System
TSG	Technical Sub-Group. A specialized sub-group within a Khronos Working Group (WG).
VR	Virtual Reality
WG	Working Group. An organized group of people working to define/augment an API.
XR	VR + AR + MR

Dedication (Informative)

In memory of Johannes van Waveren: a loving father, husband, son, brother, colleague, and dear friend.

Johannes, known to his friends as "JP", had a great sense of humor, fierce loyalty, intense drive, a love of rainbow unicorns, and deep disdain for processed American cheese. Perhaps most distinguishing of all, though, was his love of technology and his extraordinary technical ability.

JP's love of technology started at an early age --- instead of working on his homework, he built train sets, hovercrafts, and complex erector sets from scratch; fashioned a tool for grabbing loose change out of street grates; and played computer games. The passion for computer games continued at Delft University of Technology, where, armed with a T1 internet connection and sheer talent, he regularly destroyed his foes in arena matches without being seen, earning him the moniker "MrElusive". During this time, he wrote the Gladiator-bot AI, which earned him acclaim in the community and led directly to a job at the iconic American computer game company, id Software. From there, he quickly became an expert in every system he touched, contributing significantly to every facet of the technology: AI, path navigation, networking, skeletal animation, virtual texturing, advanced rendering, and physics. He became a master of all. He famously owned more lines of code than anyone else, but he was also a generous mentor, helping junior developers hone their skills and make their own contributions.

When the chance to work in the VR industry arose, he saw it as an opportunity to help shape the future. Having never worked on VR hardware did not phase him; he quickly became a top expert in the field. Many of his contributions directly moved the industry forward, most recently his work on asynchronous timewarp and open-standards development.

Time was not on his side. Even in his final days, JP worked tirelessly on the initial proposal for this specification. The treatments he had undergone took a tremendous physical toll, but he continued to work because of his love of technology, his dedication to the craft, and his desire to get OpenXR started on a solid footing. His focus was unwavering.

His proposal was unofficially adopted several days before his passing - and upon hearing, he mustered the energy for a smile. While it was his great dream to see this process through, he would be proud of the spirit of cooperation, passion, and dedication of the industry peers who took up the torch to drive this specification to completion.

JP lived a life full of accomplishment, as evidenced by many publications, credits, awards, and nominations where you will find his name. A less obvious accomplishment --- but of equal importance --- is the influence he had on people through his passionate leadership. He strove for excellence in everything that he did. He was always excited to talk about technology and share the discoveries made while working through complex problems. He created excitement and interest around engineering and technical excellence. He was a mentor and teacher who inspired those who knew him and many continue to benefit from his hard work and generosity.

JP was a rare gem; fantastically brilliant intellectually, but also warm, compassionate, generous, humble, and funny. Those of us lucky enough to have crossed paths with him knew what a privilege and great honor it was to know him. He is certainly missed.

Contributors (Informative)

OpenXR is the result of contributions from many people and companies participating in the Khronos OpenXR Working Group. Members of the Working Group, including the company that they represented at the time of their most recent contribution, are listed below.

Contributors to OpenXR 1.0

- Adam Gousetis, Google
- Alex Turner, Microsoft
- Andreas Loeve Selvik, Arm
- Andres Rodriguez, Valve Software
- Armelle Laine, Qualcomm Technologies, Inc
- Attila Maczak, CTRL-labs
- Blake Taylor, Magic Leap
- Brad Grantham, Google
- Brandon Jones, Google
- Brent E. Insko, Intel
- Brent Wilson, Microsoft
- Bryce Hutchings, Microsoft
- Cass Everitt, Facebook
- Charles Egenbacher, Epic Games
- Chris Osborn, CTRL-labs
- Christine Perey, Perey Research & Consulting
- Christoph Haag, Collabora
- Craig Donner, Google
- Dan Ginsburg, Valve Software
- Dave Houlton, LunarG
- Dave Shreiner, Unity Technologies
- Denny Rönngren, Tobii
- Dmitriy Vasilev, Samsung
- Doug Twileager, ZSpace
- Ed Hutchins, Facebook
- Gloria Kennickell, Facebook

- Gregory Greeby, AMD
- Guodong Chen, Huawei
- Jakob Bornecrantz, Collabora
- Jared Cheshier, PlutoVR
- Javier Martinez, Intel
- Jeff Bellinghausen, Valve Software
- Jiehua Guo, Huawei
- Joe Ludwig, Valve Software
- Johannes van Waveren, Facebook
- Jon Leech, Khronos
- Jonathan Wright, Facebook
- Juan Wee, Samsung
- Jules Blok, Epic Games
- Karl Schultz, LunarG
- Kaye Mason, Google
- Krzysztof Kosiński, Google
- Lachlan Ford, Microsoft
- Lubosz Sarnecki, Collabora
- Mark Young, LunarG
- Martin Renschler, Qualcomm Technologies, Inc.
- Matias Koskela, Tampere University of Technology
- Matt Wash, Arm
- Mattias Brand, Tobii
- Mattias O. Karlsson, Tobii
- Michael Gatson, Dell
- Minmin Gong, Microsoft
- Mitch Singer, AMD
- Nell Waliczek, Microsoft
- Nick Whiting, Epic Games
- Nigel Williams, Sony
- Paul Pedriana, Facebook
- Paulo F. Gomes, Samsung Electronics

- Peter Kuhn, Unity Technologies
- Peter Peterson, HP Inc.
- Philippe Harscoet, Samsung Electronics
- Pierre-Loup Griffais, Valve Software
- Rajeev Gupta, Sony
- Remi Arnaud, Starbreeze
- Remy Zimmerman, Logitech
- River Gillis, Google
- Robert Memmott, Facebook
- Robert Menzel, NVIDIA
- Robert Simpson, Qualcomm Technologies, Inc.
- Robin Bourianes, Starbreeze
- Ryan A. Pavlik, Collabora
- Ryan Vance, Epic Games
- Sam Martin, Arm
- Satish Salian, NVIDIA
- Scott Flynn, Unity Technologies
- Sean Payne, CTRL-labs
- Sophia Baldonado, PlutoVR
- Steve Smith, Epic Games
- Sungye Kim, Intel
- Tom Flynn, Samsung
- Trevor F. Smith, Mozilla
- Vivek Viswanathan, Dell
- Yin Li, Microsoft
- Yuval Boger, Sensics
- Zheng Qin, Microsoft

Index

A

xrAcquireSwapchainImage (function), 146
XrAction (type), 185
XrActionCreateInfo (type), 186
XrActionSet (type), 181
XrActionSetCreateInfo (type), 182
XrActionSpaceCreateInfo (type), 100
XrActionsSyncInfo (type), 217
XrActionStateBoolean (type), 204
XrActionStateFloat (type), 206
XrActionStateGetInfo (type), 201
XrActionStatePose (type), 211
XrActionStateVector2f (type), 208
XrActionSuggestedBinding (type), 195
XrActionType (type), 189
XrActiveActionSet (type), 218
XrAndroidSurfaceSwapchainCreateInfoFB (type), 419
XrAndroidSurfaceSwapchainFlagBitsFB (type), 418
XrAndroidSurfaceSwapchainFlagsFB (type), 418
XrAndroidThreadTypeKHR (type), 233
XrApiLayerProperties (type), 48
XrApplicationInfo (type), 55
xrApplyHapticFeedback (function), 212
xrAttachSessionActionSets (function), 195

B

XrBaseInStructure (type), 15
XrBaseOutStructure (type), 15
xrBeginFrame (function), 161
xrBeginSession (function), 125
XrBindingModificationBaseHeaderKHR (type), 237
XrBindingModificationsKHR (type), 237
XrBool32 (type), 36
XrBoundSourcesForActionEnumerateInfo (type), 221

C

XrColor4f (type), 28
XrColorSpaceFB (type), 421
XrCompositionLayerBaseHeader (type), 168
XrCompositionLayerColorScaleBiasKHR (type), 239
XrCompositionLayerCubeKHR (type), 242
XrCompositionLayerCylinderKHR (type), 246
XrCompositionLayerDepthInfoKHR (type), 249
XrCompositionLayerDepthTestVARJO (type), 521
XrCompositionLayerEquirect2KHR (type), 256
XrCompositionLayerEquirectKHR (type), 253
XrCompositionLayerFlagBits (type), 167
XrCompositionLayerFlags (type), 547
XrCompositionLayerProjection (type), 170
XrCompositionLayerProjectionView (type), 171
XrCompositionLayerQuad (type), 172
XrControllerModelKeyMSFT (type), 449
XrControllerModelKeyStateMSFT (type), 448
XrControllerModelNodePropertiesMSFT (type), 454
XrControllerModelNodeStateMSFT (type), 457
XrControllerModelPropertiesMSFT (type), 453
XrControllerModelStateMSFT (type), 457
xrConvertTimespecTimeToTimeKHR (function), 258
xrConvertTimeToTimespecTimeKHR (function), 259
xrConvertTimeToWin32PerformanceCounterKHR (function), 333
xrConvertWin32PerformanceCounterToTimeKHR (function), 332
xrCreateAction (function), 185
xrCreateActionSet (function), 181
xrCreateActionSpace (function), 99
xrCreateDebugUtilsMessengerEXT (function), 354
xrCreateHandMeshSpaceMSFT (function), 465
xrCreateHandTrackerEXT (function), 378
xrCreateInstance (function), 52
xrCreateReferenceSpace (function), 97
xrCreateSession (function), 122
xrCreateSpatialAnchorFromPerceptionAnchorMSFT (function), 487
xrCreateSpatialAnchorMSFT (function), 503
xrCreateSpatialAnchorSpaceMSFT (function), 505

xrCreateSpatialGraphNodeSpaceMSFT (function), 509
xrCreateSwapchain (function), 138
xrCreateSwapchainAndroidSurfaceKHR (function), 230
xrCreateVulkanDeviceKHR (function), 320
xrCreateVulkanInstanceKHR (function), 316
XR_CURRENT_API_VERSION (define), 548

D

XrDebugUtilsLabelEXT (type), 349
XrDebugUtilsMessageSeverityFlagBitsEXT (type), 347
XrDebugUtilsMessageSeverityFlagsEXT (type), 347
XrDebugUtilsMessageTypeFlagBitsEXT (type), 347
XrDebugUtilsMessageTypeFlagsEXT (type), 347
XrDebugUtilsMessengerCallbackDataEXT (type), 349
XrDebugUtilsMessengerCreateInfoEXT (type), 351
XrDebugUtilsMessengerEXT (type), 346
XrDebugUtilsObjectNameInfoEXT (type), 348
XR_DEFINE_ATOM (define), 551
XR_DEFINE_HANDLE (define), 550
xrDestroyAction (function), 189
xrDestroyActionSet (function), 184
xrDestroyDebugUtilsMessengerEXT (function), 356
xrDestroyHandTrackerEXT (function), 381
xrDestroyInstance (function), 56
xrDestroySession (function), 124
xrDestroySpace (function), 101
xrDestroySpatialAnchorMSFT (function), 507
xrDestroySwapchain (function), 142
XrDuration (type), 27

E

xrEndFrame (function), 163
xrEndSession (function), 128
xrEnumerateApiLayerProperties (function), 47
xrEnumerateBoundSourcesForAction (function), 219
xrEnumerateColorSpacesFB (function), 424
xrEnumerateDisplayRefreshRatesFB (function), 429
xrEnumerateEnvironmentBlendModes (function), 175
xrEnumerateInstanceExtensionProperties (function), 49
xrEnumerateReferenceSpaces (function), 95
xrEnumerateSwapchainFormats (function), 135
xrEnumerateSwapchainImages (function), 143
xrEnumerateViewConfigurations (function), 111
xrEnumerateViewConfigurationViews (function), 115
XrEnvironmentBlendMode (type), 177
XrEventDataBaseHeader (type), 39
XrEventDataBuffer (type), 39
XrEventDataDisplayRefreshRateChangedFB (type), 429
XrEventDataEventsLost (type), 40
XrEventDataInstanceLossPending (type), 59
XrEventDataInteractionProfileChanged (type), 191
XrEventDataMainSessionVisibilityChangedEXTX (type), 538
XrEventDataPerfSettingsEXT (type), 405
XrEventDataReferenceSpaceChangePending (type), 93
XrEventDataSessionStateChanged (type), 131
XrEventDataVisibilityMaskChangedKHR (type), 297
XrExtensionProperties (type), 51
XrExtent2Df (type), 33
XrExtent2Di (type), 33
XrEyeGazeSampleTimeEXT (type), 370
XrEyeVisibility (type), 174

F

XR_FAILED (define), 22
XrFlags64 (type), 13
XrFormFactor (type), 63
XrFoveatedViewConfigurationViewVARJO (type), 527
XrFovf (type), 35
XrFrameBeginInfo (type), 162
XrFrameEndInfo (type), 165
XrFrameState (type), 160
XrFrameWaitInfo (type), 159
XR_FREQUENCY_UNSPECIFIED (define), 215

G

xrGetActionStateBoolean (function), 203
xrGetActionStateFloat (function), 205
xrGetActionStatePose (function), 209
xrGetActionStateVector2f (function), 207
xrGetControllerModelKeyMSFT (function), 447
xrGetControllerModelPropertiesMSFT (function), 452
xrGetControllerModelStateMSFT (function), 455
xrGetCurrentInteractionProfile (function), 197
xrGetD3D11GraphicsRequirementsKHR (function), 265
xrGetD3D12GraphicsRequirementsKHR (function), 271
xrGetDisplayRefreshRateFB (function), 431
xrGetInputSourceLocalizedName (function), 222
xrGetInstanceProcAddr (function), 42
xrGetInstanceProperties (function), 57
xrGetOpenGLESGraphicsRequirementsKHR (function), 292
xrGetOpenGLGraphicsRequirementsKHR (function), 286
xrGetReferenceSpaceBoundsRect (function), 92
xrGetSystem (function), 64
xrGetSystemProperties (function), 67
xrGetViewConfigurationProperties (function), 113
xrGetVisibilityMaskKHR (function), 298
xrGetVulkanDeviceExtensionsKHR (function), 311
xrGetVulkanGraphicsDevice2KHR (function), 319
xrGetVulkanGraphicsDeviceKHR (function), 308
xrGetVulkanGraphicsRequirements2KHR (function), 314
xrGetVulkanGraphicsRequirementsKHR (function), 307
xrGetVulkanInstanceExtensionsKHR (function), 309
XrGraphicsBindingD3D11KHR (type), 262
XrGraphicsBindingD3D12KHR (type), 268
XrGraphicsBindingEGLMNDX (type), 541
XrGraphicsBindingOpenGLESAndroidKHR (type), 289
XrGraphicsBindingOpenGLWaylandKHR (type), 283
XrGraphicsBindingOpenGLWin32KHR (type), 279
XrGraphicsBindingOpenGLXcbKHR (type), 282
XrGraphicsBindingOpenGLXlibKHR (type), 280
XrGraphicsBindingVulkan2KHR (type), 323
XrGraphicsBindingVulkanKHR (type), 303
XrGraphicsRequirementsD3D11KHR (type), 264
XrGraphicsRequirementsD3D12KHR (type), 270
XrGraphicsRequirementsOpenGLESKHR (type), 291
XrGraphicsRequirementsOpenGLKHR (type), 285
XrGraphicsRequirementsVulkan2KHR (type), 315
XrGraphicsRequirementsVulkanKHR (type), 306

H

XR_HAND_JOINT_COUNT_EXT (define), 393
XrHandEXT (type), 380
XrHandJointEXT (type), 391
XrHandJointLocationEXT (type), 386
XrHandJointLocationsEXT (type), 384
XrHandJointSetEXT (type), 381
XrHandJointsLocateInfoEXT (type), 383
XrHandJointsMotionRangeEXT (type), 374
XrHandJointsMotionRangeInfoEXT (type), 375
XrHandJointVelocitiesEXT (type), 387
XrHandJointVelocityEXT (type), 388
XrHandMeshIndexBufferMSFT (type), 472
XrHandMeshMSFT (type), 470
XrHandMeshSpaceCreateInfoMSFT (type), 467
XrHandMeshUpdateInfoMSFT (type), 469
XrHandMeshVertexBufferMSFT (type), 473
XrHandMeshVertexMSFT (type), 475
XrHandPoseTypeInfoMSFT (type), 478
XrHandPoseTypeMSFT (type), 478
XrHandTrackerCreateInfoEXT (type), 380
XrHandTrackerEXT (type), 378
XrHapticActionInfo (type), 202
XrHapticBaseHeader (type), 213
XrHapticVibration (type), 214
XrHolographicWindowAttachmentMSFT (type), 483

I

XR_INFINITE_DURATION (define), 28
xrInitializeLoaderKHR (function), 275
XrInputSourceLocalizedNameFlagBits (type), 224
XrInputSourceLocalizedNameFlags (type), 547

XrInputSourceLocalizedNameGetInfo (type), 223
XrInstance (type), 46
XrInstanceCreateFlagBits (type), 54
XrInstanceCreateFlags (type), 547
XrInstanceCreateInfo (type), 53
XrInstanceCreateInfoAndroidKHR (type), 228
XrInstanceProperties (type), 58
XrInteractionProfileAnalogThresholdVALVE (type), 518
XrInteractionProfileState (type), 199
XrInteractionProfileSuggestedBinding (type), 194

L

xrLoadControllerModelMSFT (function), 450
XrLoaderInitInfoAndroidKHR (type), 276
XrLoaderInitInfoBaseHeaderKHR (type), 274
xrLocateHandJointsEXT (function), 382
xrLocateSpace (function), 102
xrLocateViews (function), 152

M

XR_MAKE_VERSION (define), 549
XR_MAX_EVENT_DATA_SIZE (define), 40
XR_MAY_ALIAS (define), 11
XR_MIN_HAPTIC_DURATION (define), 215

N

XR_NO_DURATION (define), 28
XR_NULL_CONTROLLER_MODEL_KEY_MSFT (define), 449
XR_NULL_HANDLE (define), 550
XR_NULL_PATH (define), 73
XR_NULL_SYSTEM_ID (define), 64

O

XrObjectType (type), 24
XrOffset2Df (type), 32
XrOffset2Di (type), 33
XrOverlayMainSessionFlagBitsEXTX (type), 537
XrOverlayMainSessionFlagsEXTX (type), 537
XrOverlaySessionCreateFlagBitsEXTX (type), 537
XrOverlaySessionCreateFlagsEXTX (type), 537

P

XrPath (type), 71
xrPathToString (function), 75
XrPerfSettingsDomainEXT (type), 397
XrPerfSettingsLevelEXT (type), 397
XrPerfSettingsNotificationLevelEXT (type), 406
xrPerfSettingsSetPerformanceLevelEXT (function), 403
XrPerfSettingsSubDomainEXT (type), 406
PFN_xrDebugUtilsMessengerCallbackEXT, 361
PFN_xrVoidFunction, 45
xrPollEvent (function), 37
XrPosef (type), 31

Q

XrQuaternionf (type), 31

R

XrRect2Df (type), 34
XrRect2Di (type), 34
XrReferenceSpaceCreateInfo (type), 98
XrReferenceSpaceType (type), 89
xrReleaseSwapchainImage (function), 150
xrRequestDisplayRefreshRateFB (function), 432
xrRequestExitSession (function), 129
XrResult (type), 17
xrResultToString (function), 60

S

XrSecondaryViewConfigurationFrameEndInfoMSFT (type), 498
XrSecondaryViewConfigurationFrameStateMSFT (type), 495
XrSecondaryViewConfigurationLayerInfoMSFT (type), 499
XrSecondaryViewConfigurationSessionBeginInfoMSFT (type), 494
XrSecondaryViewConfigurationStateMSFT (type), 496
XrSecondaryViewConfigurationSwapchainCreateInfoMSFT (type), 492
XrSession (type), 121
XrSessionActionSetsAttachInfo (type), 197
xrSessionBeginDebugUtilsLabelRegionEXT

(function), 358
XrSessionBeginInfo (type), 127
XrSessionCreateFlagBits (type), 124
XrSessionCreateFlags (type), 547
XrSessionCreateInfo (type), 123
XrSessionCreateInfoOverlayEXTX (type), 537
xrSessionEndDebugUtilsLabelRegionEXT (function), 359
xrSessionInsertDebugUtilsLabelEXT (function), 360
XrSessionState (type), 131
xrSetAndroidApplicationThreadKHR (function), 234
xrSetColorSpaceFB (function), 426
xrSetDebugUtilsObjectNameEXT (function), 353
xrSetEnvironmentDepthEstimationVARJO (function), 524
xrSetInputDeviceActiveEXT (function), 336
xrSetInputDeviceLocationEXT (function), 341
xrSetInputDeviceStateBoolEXT (function), 337
xrSetInputDeviceStateFloatEXT (function), 338
xrSetInputDeviceStateVector2fEXT (function), 339
XrSpace (type), 88
XrSpaceLocation (type), 105
XrSpaceLocationFlagBits (type), 106
XrSpaceLocationFlags (type), 547
XrSpaceVelocity (type), 107
XrSpaceVelocityFlagBits (type), 108
XrSpaceVelocityFlags (type), 547
XrSpatialAnchorCreateInfoMSFT (type), 504
XrSpatialAnchorMSFT (type), 502
XrSpatialAnchorSpaceCreateInfoMSFT (type), 506
XrSpatialGraphNodeSpaceCreateInfoMSFT (type), 511
XrSpatialGraphNodeTypeMSFT (type), 511
xrStopHapticFeedback (function), 215
xrStringToPath (function), 73
XrStructureType (type), 544
xrStructureTypeToString (function), 61
xrSubmitDebugUtilsMessageEXT (function), 357
XR_SUCCEEDED (define), 22
xrSuggestInteractionProfileBindings (function), 192
XrSwapchain (type), 135
XrSwapchainCreateFlagBits (type), 142
XrSwapchainCreateFlags (type), 548
XrSwapchainCreateInfo (type), 139
XrSwapchainImageAcquireInfo (type), 147
XrSwapchainImageBaseHeader (type), 145
XrSwapchainImageD3D11KHR (type), 263
XrSwapchainImageD3D12KHR (type), 269
XrSwapchainImageOpenGLESKHR (type), 290
XrSwapchainImageOpenGLKHR (type), 284
XrSwapchainImageReleaseInfo (type), 151
XrSwapchainImageVulkan2KHR (type), 325
XrSwapchainImageVulkanKHR (type), 305
XrSwapchainImageWaitInfo (type), 149
XrSwapchainStateAndroidSurfaceDimensionsFB (type), 435
XrSwapchainStateBaseHeaderFB (type), 435
XrSwapchainStateSamplerOpenGLESFB (type), 436
XrSwapchainSubImage (type), 169
XrSwapchainUsageFlagBits (type), 137
XrSwapchainUsageFlags (type), 548
xrSyncActions (function), 216
XrSystemColorSpacePropertiesFB (type), 424
XrSystemEyeGazeInteractionPropertiesEXT (type), 369
XrSystemFoveatedRenderingPropertiesVARJO (type), 526
XrSystemGetInfo (type), 65
XrSystemGraphicsProperties (type), 69
XrSystemHandTrackingMeshPropertiesMSFT (type), 464
XrSystemHandTrackingPropertiesEXT (type), 377
XrSystemId (type), 64
XrSystemProperties (type), 68
XrSystemTrackingProperties (type), 70

T

xrThermalGetTemperatureTrendEXT (function), 409
XrTime (type), 26

xrTryGetPerceptionAnchorFromSpatialAnchorMSFT (function), 488

U

XR_UNQUALIFIED_SUCCESS (define), 22
xrUpdateHandMeshMSFT (function), 468
xrUpdateSwapchainFB (function), 438

V

XrVector2f (type), 29
XrVector3f (type), 30
XrVector4f (type), 30
XrVersion (type), 5
XR_VERSION_MAJOR (define), 549
XR_VERSION_MINOR (define), 549
XR_VERSION_PATCH (define), 550
XrView (type), 154
XrViewConfigurationDepthRangeEXT (type), 412
XrViewConfigurationProperties (type), 114
XrViewConfigurationType (type), 110
XrViewConfigurationView (type), 117
XrViewConfigurationViewFovEPIC (type), 416
XrViewLocateFoveatedRenderingVARJO (type), 529
XrViewLocateInfo (type), 154
XrViewState (type), 155
XrViewStateFlagBits (type), 156
XrViewStateFlags (type), 548
XrVisibilityMaskKHR (type), 296
XrVisibilityMaskTypeKHR (type), 295
XrVulkanDeviceCreateFlagsKHR (type), 322
XrVulkanDeviceCreateInfoKHR (type), 321
XrVulkanGraphicsDeviceGetInfoKHR (type), 320
XrVulkanInstanceCreateFlagsKHR (type), 318
XrVulkanInstanceCreateInfoKHR (type), 317
XrVulkanSwapchainFormatListCreateInfoKHR (type), 330

W

xrWaitFrame (function), 157
xrWaitSwapchainImage (function), 148