Future Proof XR

Lachlan Ford
Software Engineer, Microsoft
OpenXR Working Group Member
September 2020

First of all, thank you!

Agenda

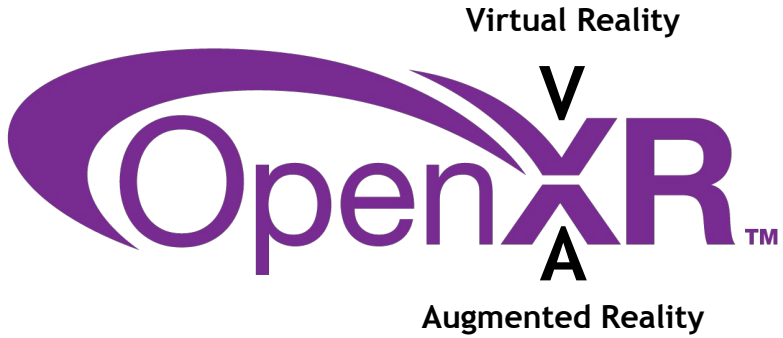
- About Me
- Why am I talking today?
- Future proofing, why and how.
- Questions and Answers (if applicable)

About Me: Lachlan Ford

- **Software Engineer on the Microsoft Mixed Reality team.**
  - Focus on helping to implement open standards such as OpenXR and WebXR.
- **OpenXR working group participant**
  - Focus on helping to define the OpenXR standard since early 2018.
- **Have been programming since forever**

Why “X”R?

Why “X”R?



Why “X”R?

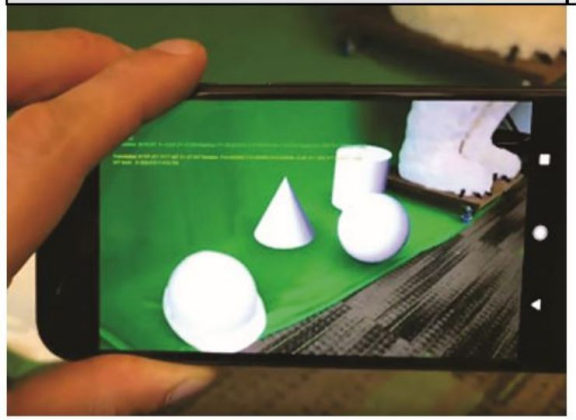
Camera Passthrough AR	Stereoscopic VR / AR	Projection CAVE
		

Photo Credit: Dave Pape

- **Many other forms including:**
  - Pass through stereoscopic
  - Handheld transparent
  - Magic window systems (handheld or static)
  - Other unorthodox configurations yet to be built!

Why “X”R?

This was the motivation behind many of our decisions in designing the API.

OpenXR is designed to let you build apps that run on many XR systems, leveraging their unique value, with the same code.

But this does not come for free!

Future Proofing

- **Many API tools at our disposal**
  - Form Factor
  - View Configuration
  - Environment Blend Modes
  - Session State Lifecycle
  - Spaces
  - View Projection Location
  - Frame Timing
  - Interaction Profiles and Actions
  - Extensions

Future Proofing

- **Form Factor**
  - Form factor restricts support to the class of system.
    - e.g. HMD vs Phone
    - Systems may support multiple form factors
      - e.g. XR cabale phones support passthrough AR **AND** head mounted VR
    - Important because the form factor informs the UX
    - Form factor support is explicit and opt-in
    - Querying the runtime for its support allows an app to dynamically choose its UX and behavior

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

Future Proofing

- **View Configuration**
  - Each system has its own ways to render
  - There are general classes of view configurations (e.g. mono, stereo) and more special kinds of view configurations (e.g. _QUAD_VARJO)
    - Special view configurations usually require special handling
  - You **cannot** assume a view configuration based on form factor.
    - e.g. An HMD may **NOT** support stereo view configuration
  - Querying the runtime for its support allows an app to render correctly to all kinds of hardware

```
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;
```

Future Proofing

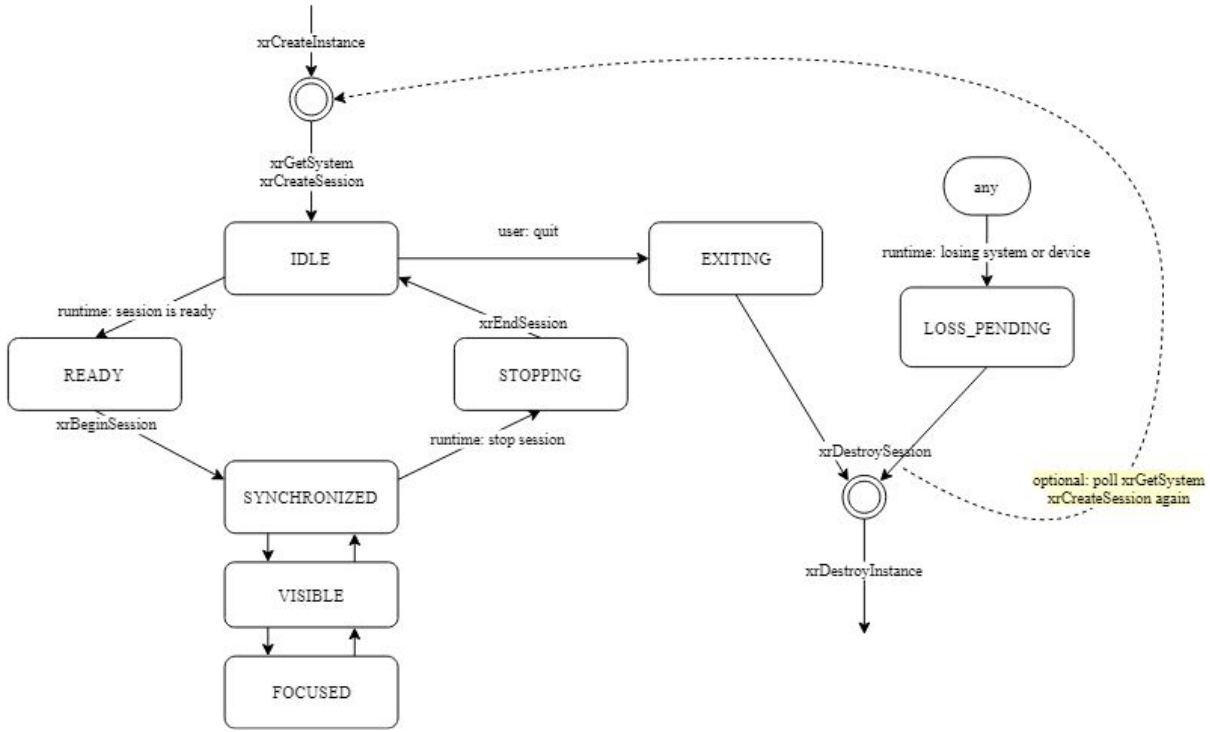
- **Blend modes**
  - For the best experience you will want to prepare and render your assets based on the blend mode that the runtime supports
  - **Cannot** assume that the blend mode matches the form factor
    - e.g. passthrough VR will be alpha blended in passthrough mode but opaque otherwise.

```
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;
```

```
XrResult xrEnumerateEnvironmentBlendModes(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    XrViewConfigurationType                     viewConfigurationType,
    uint32_t                                    environmentBlendModeCapacityInput,
    uint32_t*                                   environmentBlendModeCountOutput,
    XrEnvironmentBlendMode*                     environmentBlendModes);
```

Future Proofing

- **Session state lifecycle**
  - An abstract session lifetime designed to be platform agnostic
  - Some hardware may be more or less strict about this but it should be respected either way



Future Proofing

- **Spaces**
  - Not all reference spaces are supported. You will need to check this.
    - e.g. XR_REFERENCE_SPACE_TYPE_STAGE is currently unsupported on HoloLens
  - View space is **NOT** for finding the position of the head
    - It will not be near the head position on all runtimes
    - Assuming this may be approximately correct on some runtimes but completely wrong on other runtimes
    - View space is useful for targeting rays and screen space / view locked positioning of UI elements.

```
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_MAX_ENUM = 0x7FFFFFFF
} XrReferenceSpaceType;
```

Future Proofing

- **View / Projection querying**
  - xrLocateViews is what you **ALWAYS** want to use to query the view / projection data for the system you are currently rendering to
  - Attempting to manipulate the data (view, fov, projections) yourself may work on some systems but will fail catastrophically on other systems
    - Some systems allow this through explicit means
      - e.g. XR_EPIC_view_configuration_fov allows you to modify the FOV explicitly

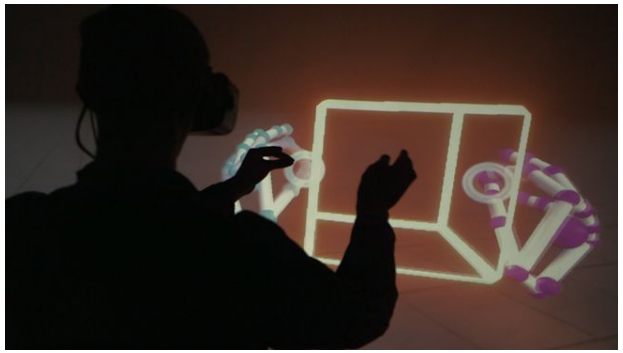
```
XrResult xrLocateViews(
    XrSession                                   session,
    const XrViewLocateInfo*                     viewLocateInfo,
    XrViewState*                                viewState,
    uint32_t                                    viewCapacityInput,
    uint32_t*            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

Future Proofing

- **Interaction Profiles and the Action system**
  - Design from the ground up for future proofing!
  - Input mechanisms are rapidly evolving and ideally apps can achieve forwards and backwards compatibility for free by leveraging the OpenXR action system correctly
  - Interaction profiles are **NOT** devices
    - You cannot presume to know the system you are running based on the current interaction profile
    - The current interaction profile **should** inform the UX of the app
  - If used correctly your app could run on all manner of future hardware like brain computer interfaces when they are common!

Future Proofing

- **Extensions**
  - When all the foundations are in place, extensions allow you to target emerging XR capabilities that need their own APIs not covered by the core OpenXR API
  - Some extensions are multi-vendor (EXT, KHR) and some extensions are single-vendor
    - The presence of single-vendor extensions does not imply you are running on a specific system and this cannot be assumed.



Conclusion

- **OpenXR is a platform agnostic API**
  - We attempt to encapsulate as much platform eccentricities and specifics inside the runtime implementations
  - Most of the API is general, but some specific things are required to be handled
    - e.g. Form factor, view configuration and blend mode for UX and rendering
  - Utilizing the API correctly will go a long way towards future proofing your app, allowing it to run on as much past and future hardware as possible, drastically increases its platform compatibility and decreasing its maintenance cost, by offloading platform specific support to runtimes
- **Do not try to outsmart the API**
  - Fingerprinting is bad
    - Your app will **NOT** be future proof
  - Do not make assumptions about the properties of the hardware you're running on
    - It may work on some hardware configurations and catastrophically fail on others!
    - **Trust the runtime** (or give the runtime enough information to do a good job!)
  - #ifdef PLATFORM is a smell that we want to minimize

Thank you! 谢谢

Lachlan Ford
Software Engineer, Microsoft
OpenXR Working Group Member
September 2020