THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

OpenXR and the Monado Runtime: A Standardized (and Step-by-Step) Solution to AR/VR Fragmentation

ANASTASIA SEREMULA
SPRING 2022

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Computer Science
with honors in Computer Science

Reviewed and approved* by the following:

Anand Sivasubramaniam
Distinguished Professor of Computer Science and Engineering
Thesis Supervisor

Rebecca Passonneau
Professor of Computer Science and Engineering
Honors Adviser

* Electronic approvals are on file.

ABSTRACT

The rapid expansion of augmented reality (AR) and virtual reality (VR) continue to shape how individuals communicate and interact with computers in both their personal lives and numerous sectors in society. From healthcare and education to manufacturing and construction, an increasing amount of technology companies (such as Microsoft, Meta/Facebook, and Google) are investing in proprietary hardware systems to support and profitize on their AR/VR visions. However, with numerous platforms (all following different specifications) now available for consumers, developers, and hardware vendors to choose from, this damaging competition in the AR/VR industry is causing the market to become fragmented. Due to this, developers must painstakingly port or re-write their code multiple times in order to be compatible with several devices. In this thesis, we explore the steps needed to write an AR/VR application for OpenXR and one of its emerging runtime environments, Monado, that aim to solve this portability problem by providing a standardized and open source API that all hardware systems can soon follow. Allowing true cross-platform experiences, developers can then focus on AR/VR product development that is guaranteed to be deployable on any OpenXR-compatible hardware with no additional hassle.

TABLE OF CONTENTS

LIST OF FIGURES iii

LIST OF TABLES iv

ACKNOWLEDGEMENTS v

Chapter 1: Introduction 1

1.1 Motivation 1
1.2 Thesis Outline 2

Chapter 2: Background 3

2.1 Notable AR/VR Hardware Devices 3
2.1.1 Oculus Quest 2 (Meta Quest 2) 3
2.1.2 Microsoft HoloLens 2 4
2.1.3 Google Cardboard 6
2.2 Programming Environments Available for AR/VR Hardware Devices 7
2.3 A General Overview of OpenXR and Monado 9

Chapter 3: Writing OpenXR Programs with Monado Compatibility 12

3.1 Architecture of OpenXR 12
3.2 Step 1: Creating an Instance 13
3.3 Step 2: Determining Where and How to Run 14
3.4 Step 3: Establishing Interaction and Input Handles 15
3.5 Step 4: Preparing the Immersive Experience 18
3.5.1 Creating a Session 18
3.5.2 Attaching Action Sets 19
3.5.3 Building Reference and Action Spaces 19
3.5.4 Constructing Swapchains 21
3.6 Step 5: Participating in the Frame Loop 22
3.6.1 Handling Input 24
3.6.2 Polling Events 25
3.7 Summary of Steps 26

Chapter 4: Conclusion 28

4.1 Key Takeaways 28
4.2 Future Work 29

BIBLIOGRAPHY 31

LIST OF FIGURES

Figure 1 (OpenXR's Unifying Structure) 11

Figure 2 (Functions and Structures for Step 1) 14

Figure 3 (Functions for Step 2) 15

Figure 4 (Functions for Step 3) 17

Figure 5 (Functions and Structures for Step 4 - Sessions) 18

Figure 6 (Functions for Step 4 – Building Reference and Action Spaces) 20

Figure 7 (Functions for Step 4 - Swapchains) 21

Figure 8 (Functions and Structures for Step 5 – Frame Loop) 23

Figure 9 (Functions for Step 5 - Swapchains) 24

Figure 10 (Functions for Step 5 - Input) 25

Figure 11 (Functions and Structures for Step 5 – Event Polling) 26

Figure 12 (OpenXR Hierarchy) 27

LIST OF TABLES

Table 1 (Example: Interaction Profile Bindings for Oculus Touch Controller) 17

ACKNOWLEDGEMENTS

I am very fortunate to have been surrounded by amazing people while creating this thesis. I would like to thank these people, without whom I would not have been able to complete this honors thesis!

First, I would like to extend my deepest gratitude to my Thesis Supervisor, Dr. Anand Sivasubramaniam, for providing me the opportunity to conduct research for the past year in his lab. Moreover, I want to thank him for his exemplary guidance and insightful advice throughout all stages of this research project.

Second, I would like to extend my sincere thanks to Dr. Shulin Zhao and the members of Penn State's Computer Systems Lab for providing their precious time and effort. The completion of my undergraduate thesis would not have been possible without their collaborative efforts and willingness to impart their knowledge to me.

Third, I must also thank the Schreyer Honors College and College of Engineering for the numerous opportunities and once-in-a-lifetime experiences that have greatly helped me develop my interests and passions during my time at Penn State. Specifically, I very much appreciate my commonwealth campus, Penn State Lehigh Valley, for providing me with a solid foundation that I have heavily relied on to shape my academic and personal growth.

Fourth, I cannot begin to express my thanks to my Mom, Dad, and two little sisters, Tatiana and Katerina, for their unconditional love and support throughout the entire research and writing process as well as during my past four years at Penn State. Their constant motivation, direction, humor, and strength have pushed me to become the leader that I am today - I very much appreciate everything you do for me!

Chapter 1:
Introduction

1.1 Motivation

In today's world, augmented reality and virtual reality are completely revolutionizing the way individuals interact with computers. While augmented reality (AR) is a type of interactive technology that enhances physical objects and environments in the real-world with digital data, images, and other sensory stimuli, virtual reality (VR) is a type of technology that creates a simulated 3D environment, or "virtual" digital world, that can be explored and interacted with by users [1]. Though still in their infancy, these technologies and their applications are quickly receiving mainstream attention for their ability to bridge the digital and physical world, already affecting companies, universities, and social enterprises in every sector [2]. In the coming years, AR/VR will not only transform how individuals learn, make decisions, and interact with the physical world, but also shake up how global industries create value and reshape competition [2]. Moreover, from immersing individuals in alternative universes to boosting the raw processing power in mobile chips and networks, AR/VR is becoming an increasingly important tool to reinvent the personal computing experience. With pioneering organizations, such as Amazon, General Electric, and the US Navy, already witnessing AR/VR's quality and productivity gains, these technologies will change the future, especially in diverse industries like healthcare, sports, education, and manufacturing, by allowing users to express and explore the many complexities behind the human experience [2].

In order to accomplish this vision, AR/VR cannot just be used as it is provided; it is crucial that developers and content creators not only have access to the tools needed for building AR/VR applications, but also know how to write and customize programs for these technologies to meet society's different kinds of needs. This is because programming for AR/VR applications involves writing code that successfully handles real-world interactions, events, and objects, a departure from traditional coding practices that mainly rely on writing efficient straight-line code or algorithms. Due to the new style of thinking necessary for writing AR/VR programs, this challenge is currently a very important and popular focus for the technology's early adopters.

1.2 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 introduces notable AR/VR hardware systems in today's market, including their specifications, limitations, and what can be developed with each. It also describes the programming environments available for these platforms before highlighting AR/VR fragmentation in the industry, eventually providing a general overview of OpenXR and the Monado runtime. Chapter 3 takes an in-depth dive into the OpenXR Specification, exploring the key parts of an OpenXR program for AR/VR. This includes discussing the five main steps that are essential to writing an OpenXR application with Monado compatibility. Finally, Chapter 4 summarizes the findings of this thesis, shares lessons learned while developing AR/VR applications, and offers suggestions for future research.

Chapter 2:
Background

2.1 Notable AR/VR Hardware Devices

Recent advances in the power of AR/VR hardware have allowed new headsets and processors to emerge in the mainstream market. These technologies usually fall into two primary categories: standalone and tethered devices. While standalone devices integrate all the components necessary into the headset to provide VR experiences, tethered devices are headsets that display content to a different device, such as a PC or video game console, in order to provide a virtual reality experience [3]. However, mobile headsets also exist; these combine a smartphone with a mount to simulate AR/VR experiences found in standalone or tethered devices. While this list is far from expansive, notable devices today include the Oculus Quest 2 (Meta Quest 2), Microsoft HoloLens 2, and Google Cardboard. We now explore what these devices differ in, as well as what can be created with each, in the remainder of this section.

2.1.1 Oculus Quest 2 (Meta Quest 2)

Created by Meta Platforms, the Quest 2 was released as a successor to the company's previous headset, the Oculus Quest, on September 16th, 2020 [6]. With 6GB of RAM and a VR-optimized Qualcomm Snapdragon XR2 chipset, the Quest 2 aims to provide seamless and immersive experiences "on the go" through four in-headset cameras, 90Hz refresh rates, and a single LCD panel that is split to display 1832 x 1920 pixel resolution per eye [4, 6]. It also intends to be a portable and easily-accessible device for individuals of all ages 13 and up;

wireless and battery-powered for free roam, its standalone headset comes with two Oculus Touch controllers that feature hand tracking to achieve a full-motion six degrees of freedom (6DoF) [4, 6]. Acting as a platform for both new and veteran VR developers, the device also runs an Android-based operating system (utilizing source code from Android 10) and promotes a frequently-updated Oculus SDK package that includes the libraries, tools, and resources needed for native C/C++ development of Android apps [6]. Moreover, passthrough API technology in the SDK build highlights the Quest 2's AR component; the headset is able to overlay augmented reality objects and images into virtual environments for added value and functionality [7].

Ultimately, while all computing and motion-tracking is contained on the Quest 2, an iOS or Android smartphone/tablet is necessary for first-time setup. Users are also required to have a Facebook account in order to use the Quest 2 and future Oculus products, possibly steering away those unwilling to share their data. Nevertheless, the Quest 2's $299 price point and compact and lightweight design emphasizes its goal to expose more individuals to VR [5]. From providing 360-degree content to hosting numerous apps, games, and video content of all genres, the self-contained device is suitable for developers who want their products to reach large audiences [5].

2.1.2 Microsoft HoloLens 2

Created by Microsoft, the Microsoft HoloLens 2 was released as a successor to the company's previous mixed reality headset, the Microsoft HoloLens, on November 7th, 2019. With 4GB of RAM and an ARM-based Qualcomm Snapdragon 850 chipset, the HoloLens 2 aims to provide seamless holographic experiences for corporations through 6 in-headset cameras (4 of which used for positional tracking), built-in artificial intelligence (AI) tools, and an

advanced holographic processing unit (HPU) [9]. It also intends to be a portable device for workers, such as those in auto shops, factory floors, and operating rooms, who primarily work with their hands and have trouble integrating technology into their daily work; wireless and battery-powered for free roam, the standalone AR-headset comes with transparent and flip-down lenses to allow eye contact without removing the device [8, 9]. Through laser-based displays, oscillating mirrors, and cutting-edge waveguides (glass pieces carefully etched and placed in front of the eyes to reflect holograms), HoloLens 2 gives users a wider field of view (6DoF) and brighter images without the need for controllers; its new array of sensors (called Azure Kinect) track when users pinch, point, and tap on holograms with their hands like real objects [8, 10]. Moreover, the device runs a Windows-based operating system (utilizing Windows 10 Holographic) and not only promotes the Windows 10 SDK, but also a Mixed Reality Toolkit (MRTK) that includes a set of core components and scripts for developers to use as a framework for rapid prototyping, spatial interactions and UI, and cross-platform mixed reality app development [10, 11]. Microsoft is also constantly creating new software tools for HoloLens 2; for example, Dynamic 365 Guides is a mixed reality application that contains templates to create instructions for repairing real-world objects, while Azure Remote Rendering offloads compute loads from the device to the cloud in order to provide finer levels of detail [8].

Ultimately, while all computing and motion-tracking is contained on the HoloLens 2, the device is only available to enterprise customers due to it being marketed as a scalable learning and visualization tool for corporations in various sectors in society [8]. However, even with the lack of a consumer-centric, daily use application, the $3500+ device shines in providing communications and telepresence apps for business [8]. With a small and lightweight design, the HoloLens 2 allows developers to create AR software for training and collaboration in the

workplace in order to increase user accuracy and input [9]. From clean rooms to hazardous environments, developers can build industry-related and customizable processes, add-ons, or applications to digitalize business operations and help workers get work done.

2.1.3 Google Cardboard

Created by Google, Google Cardboard is a VR platform that was released on June 25th, 2014. Named after the fold-out cardboard viewer where a smartphone is placed, Google Cardboard aims to be a simple, low-cost, and easily-accessible headset for users in order to boost interest and development for VR applications [16]. By creating their own viewer with inexpensive components (such as velcro, tape, and cardboard) or purchasing a pre-manufactured one, users can place their phone into the back of the viewer and view a variety of content through the lenses after downloading Cardboard-compatible mobile apps on their device [17]. From Google Expeditions (where various destinations, landmarks, and landforms worldwide can be explored) to Sketchfab (where AR and VR combine to create functional 2D/3D animations), numerous apps, games, and videos of all genres and qualities are available on Google Play or Apple's App Store for users to wirelessly experience 360-degree (3DoF) content with, highlighting Google Cardboard's ability to work with hardware an individual already owns [13]. Due to this, Google provides three software development kits for developers: one for the Android operating system using Java, one for the game engine Unity using C#, and one for the iOS operating system [12]. The company also provides an open source Cardboard SDK package that includes essential VR features, such as motion tracking, stereoscopic rendering, and user interaction via the viewer button, that can be used to build immersive cross-platform VR

experiences on new or existing apps [12, 15]. One expansion of the package, VR View, is an open source tool (available on GitHub) that allows developers to self-host and embed 360-degree VR content on web pages or mobile apps via desktop, Android, and iOS devices [14].

While Google Cardboard's light, cheap, and goofy design allowed AR/VR experiences to be more approachable to the general public (especially to those unfamiliar with it), interest in the platform and creating apps for it have considerably declined since launch [13]. Due to this, the Google Store stopped selling cardboard viewers in March 2021 and active development on Cardboard's SDK packages is less frequent [13]. However, even though Google Cardboard cannot simulate free-roaming experiences found on proper virtual reality hardware, the platform is still consistently used today for education and entertainment purposes; it is best suited for developers interested in designing simple AR/VR experiences or aim to have their Android or iOS applications accessible to everyone [13].

2.2 Programming Environments Available for AR/VR Hardware Devices

As more standalone and tethered AR/VR hardware systems from various companies enter the mainstream market to profitize on their proprietary content, developers and hardware providers are struggling to catch up. Due to this, several programming environments, all with minimal standardization, are emerging for developing applications on these devices. For example, Unity is a cutting-edge cross-platform game engine for AR and VR creators that supports 20+ platforms and allows real-time development; featuring intuitive UI and tools, developers can easily create and test prototypes with Unity's highly extensible platform while adapting the C# scripting system, API, and source code to their own needs [18]. Unity also

optimizes graphics performance, provides rich interactivity through its physics, rendering, and communications system, and hosts a Unity Asset Store full of thousands of unique assets and productivity tools to jump-start development [18]. Meanwhile, Unreal Engine is a leading cross-platform game engine that frequently updates its system to support the diverse amount of extended reality (XR) hardware and software available today [19]. With customizable AR/VR applications via Blueprint Visual Scripting (a complete gameplay scripting system utilizing a node-based interface to define object-oriented classes or objects) or direct C++ coding, developers can efficiently add new features or build upon old ones [19, 20]. In addition to the wide array of free templates, samples, and assets available, Unreal Engine lets developers scale their applications to meet the performance requirements of high or low-power devices through its graphics pipelines, maintaining speed without harming graphic quality [19].

Providing easy-to-use and powerful development frameworks, the popularity of these two game engines has caused an increasing amount of AR/VR third-party software tools, all with different purposes and requirements, to be integrated [18]. For example, Vuforia is a popular cross-platform software development kit that handles augmented reality and mixed reality experiences on numerous hardware devices through its robust tracking and performance [21]. Using advanced computer vision techniques, the SDK can be easily integrated into Unity or Unreal Engine to provide virtual buttons, place digital objects into a real-world environment, and identify and track image targets, text, and 3D objects in real-time [22]. Moreover, the Wikitude SDK is extensively used for location-based AR and Cloud recognition, while the ARKit (an AR framework for iOS applications) and ARCore (an AR framework for Android applications) SDKs can be implemented via plugins to create handheld AR experiences [22, 23]. While not extensive, the examples presented highlight how Unity and Unreal Engine must “mix and match”

unique SDKs together before developers can build AR/VR applications for a certain hardware. However, due to the rapid evolution of AR/VR software and its programming requirements, the customizability of these SDKs is quickly leading to AR/VR fragmentation, making it difficult for developers to build seamless experiences that work on every headset without porting or rewriting their code.

2.3 A General Overview of OpenXR and Monado

As AR/VR programming environments and platforms continue to diversify, a set of standard APIs is needed to prevent future AR/VR software from being individually tailored to every hardware system available. Reasonable efforts have been made at standardizing these applications with open source: Khronos' open source working group has recently released OpenXR, an open and royalty-free standard for AR/VR platforms and devices that uses a single API to enable developers to code portable AR/VR applications that work on any OpenXR-compliant headset [24]. With OpenXR 1.0 released on July 29, 2019, the high-performance API is unique by never referring to a physical device when coding, instead relying on abstracted actions like "grab," "jump," and "point" that can be bound to an array of inputs at the hardware level [24, 30]. Handling frame composition, view re-projection, headset management, and head and hand tracking information, developers can also use any 3D graphics library of their choice since the OpenXR framework ensures that the images will look correct on any headset; this simplifies cross-platform AR/VR development and enables applications to reach numerous hardware platforms and vendors without being modified [24, 25].

With OpenXR providing an interface that handles frame composition and peripheral management between an application and an in-process or out-of-process XR runtime, an increasing amount of holographic and immersive platforms and devices are now planning to support the standard [30]. From Unity and Unreal Engine to HoloLens 2 and Quest 2, new VR headsets are also being created from Khronos member companies; for example, we study Collabora's fully open source Monado "headset" in this paper [29]. Being the first of its kind, Monado is a complete and conformant implementation of the OpenXR API for GNU/Linux [27]. With features that process non-standard input from head-mounted display (HMD) and controller devices before rendering these devices with the API, the Monado runtime is built around a Vulkan API-powered compositor that handles OpenGL and Vulkan XR applications in both extended mode and direct mode [26]. It also integrates hardware support with both its own native drivers and community-built ones (such as those from OpenHMD and libsurvive), offering the fundamental building blocks necessary to jump-start development [26]. Aiming to make GNU/Linux a fully-enabled XR platform that can reduce maintenance and development costs, Monado is supporting desktop devices (targeting VR use cases and camera-based see-through) and creating a fully open driver stack that can be integrated into several system on a chip (SoC) based mobile platforms [26]. In addition, the Monado XR runtime strives to build a community that can test and optimize their AR/VR technologies across a variety of supported devices and software implementations using 6DoF; by reducing the barrier to developing novel or consumer-centric OpenXR extensions, device drivers, and applications, Monado emphasizes its focus on innovation across the entire XR ecosystem through the open source nature of its GPU/HMD drivers and XR middleware [26, 27]. Ultimately, this allows developers to collaborate on a common code base in order to experiment with new XR technologies or easily develop and

deploy existing products.

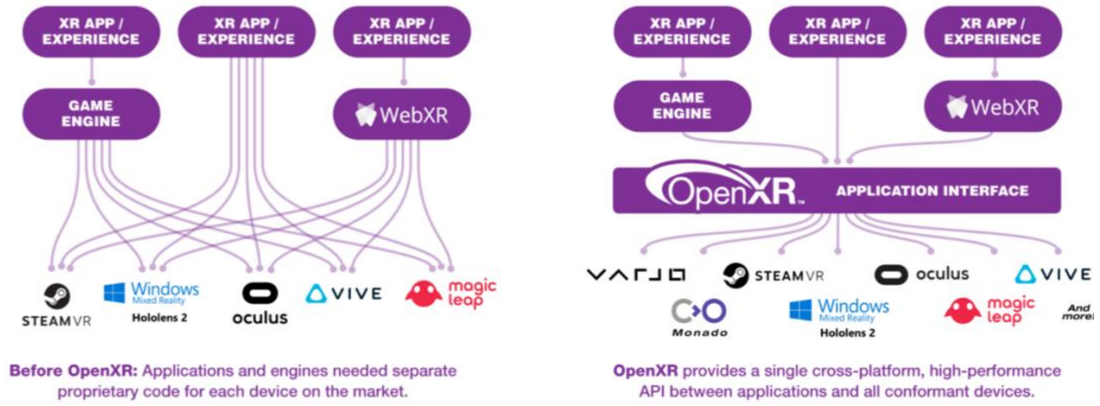

Figure 1 (OpenXR's Unifying Structure)

Chapter 3:
Writing OpenXR Programs with Monado Compatibility

3.1 Architecture of OpenXR

For developers, OpenXR contains a set of functions that can interact with a runtime like Monado to perform required operations, such as getting tracking positions, accessing controller states, and submitting rendered frames [29]. It is also composed of a variety of unique components in order to create an API that sufficiently handles AR/VR development. These include, but are not limited to:

- `XrSpace`: a variable that represents 3D space
- `XrInstance`: a variable that represents the OpenXR runtime
- `System` and `XrSystemId`: variables that represent hardware devices (such as AR/VR devices and controllers)
- `XrActions`: a variable that handles user inputs
- `XrSession`: a variable that handles the interaction session between an application and a user

It is important to note that `XrSystemId` is of type `atom`. `atom`s are a type that do not have an explicit lifetime; they are coded numbers that represent fixed elements in the Monado runtime [31]. Due to this, the most common atom used during development is an `XrPath`, which is a number that corresponds with a particular `XrInstance` to a string that represents a semantic path. Using all these components, typical OpenXR applications follow a particular structure outlined in this chapter's remaining sections.

3.2 Step 1: Creating an Instance

First, developers must create and configure an instance by selecting the desired extensions, which are able to expose new features or modify the behavior of existing functions [30]. One extension, known as a graphics binding extension, must be enabled in order to make OpenXR applications [31]. To view available extensions on a particular system, developers can utilize the `xrEnumerateInstanceExtensionProperties` function. However, if a developer is unsure of which extensions are needed or wishes to run their application with specific extensions they know how to use, the instance can just be created in order to determine which extensions are required [31].

Developers must also select their desired API layers. API layers are placed in-between a developer's application and the Monado runtime for API hooking, aiding in logging, debugging, and validation [30]. While a facility for these layers can be configured outside a developer's application via environment variables so a loader can automatically implement them, an application can also load API layers directly by enumerating the ones that are available before an instance is created [31]. This is done by utilizing the `XrApplicationInfo` structure. By adding information about the application, such as name and engine name version, to `XrApplicationInfo`, the Monado runtime can successfully identify the application. The `xrCreateInstance` function will then use this information to build an instance handle.

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

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;

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

Figure 2 (Functions and Structures for Step 1)

3.3 Step 2: Determining Where and How to Run

While OpenXR 1.0 natively supports handheld monoscopic "magic window"-style AR and stereoscopic head-mounted displays, not all runtimes support these form factors [31]. Due to this, the next step for developers is obtaining an `XrSystemId` in order to identify their hardware system. Using the `xrGetSystem` function to find their desired form factor, developers will obtain an `XrSystemId` if it is available. However, the desired form factor may be unavailable if the developer's hardware device does not support it, or temporarily unavailable if the developer's hardware device is either not plugged in or must be switched to a different mode (if the chosen hardware device has this capability) [31]. Once a system exists, view configurations (`MONO`, `STEREO`, etc.) can then be established. For developers supporting more than one view configuration, the `xrEnumerateViewConfigurations` function can be used not only to determine which ones are supported by the developer's chosen system, but also to pick the

primary view configuration that will be rendered for their application [31]. Nevertheless, regardless of which configuration is chosen, developers will eventually call `xrEnumerateViewConfigurationViews` to obtain the correct number of fixed views for their view configuration; for example, `MONO` has one view, `STEREO` has two views, and the XR `QUAD_VARJO` (a vendor extension) has four views [31]. These configurations are extensively referred to in the OpenXR Specification when developing AR/VR applications.

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

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

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

Figure 3 (Functions for Step 2)

3.4 Step 3: Establishing Interaction and Input Handles

Once a developer creates an instance, interactions can then be established through `Action`s and `ActionSet`s. `Action`s are meaningful bits of interaction that a user does in an application, such as move and jump [31]. Since OpenXR stores user actions in a developer's

application instead of on the buttons and controllers used to perform these actions, several different types of `Actions` can be created by calling `xrCreateAction`. For example, `Boolean` actions are button actions that can be toggled on or off. `Float` actions correspond to analog triggers, while `Vector2f` actions are two-dimensional `Float` actions that represent thumbsticks or trackpads. Moreover, `Pose` actions correspond to tracked objects (such as the user's hands), while `Haptic` actions are output actions that allow developers to provide rumble feedback to the user [31]. Together, these numerous actions are categorized into different `ActionSet`s, which are a group of related actions that are application-defined and correspond to a particular usage context (such as menu and gameplay) [30]. Since one or more `ActionSet`s can be active at a single point in time, OpenXR's hardware independence is maintained in order to maximize user-accessibility and simplify support for future input devices [31].

With `Action`s and `ActionSet`s established, a developer can then customize how these actions are performed on the hardware devices being used for testing and debugging. To accomplish this in OpenXR without excluding other hardware devices, each controller type has an "interaction profile" that recognizes the collection of buttons and other input sources in a physical arrangement [29]. While these profiles can be found in the OpenXR Specification for a variety of hardware devices, additional interaction profiles can also be added through extensions. For every interaction profile, a developer submits pairs of actions and suggested bindings, or parts of the controller that will be used to execute a particular action [31]. Even though developers can create as many action-binding pairs as needed, it is important to note that not all actions need suggested bindings. In a call to the `xrSuggestInteractionProfileBindings` function, multiple bindings for each action can be defined; for example, the left and right controllers could both trigger an action known as `grab_object`. An `XrPath` atom is also

utilized to represent a hierarchical string when suggesting these bindings; for example, the path string "`/user/hand/right/input/select/click`" refers to the click of a `select` button on a controller device meant to resemble the user's right hand [31]. These paths, naming conventions, and standardizations are detailed in the OpenXR Specification. Applications must provide default bindings for their actions so applications and runtimes can successfully coordinate action-to-input mapping for input data [30].

```
XrResult xrCreateAction(
    XrActionSet                                 actionSet,
    const XrActionCreateInfo*                   createInfo,
    XrAction*                                   action);
XrResult xrSuggestInteractionProfileBindings(
    XrInstance                                  instance,
    const XrInteractionProfileSuggestedBinding* suggestedBindings);
```

Figure 4 (Functions for Step 3)

Table 1 (Example: Interaction Profile Bindings for Oculus Touch Controller)

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

3.5 Step 4: Preparing the Immersive Experience

In OpenXR, assembling the AR/VR application for user functionality and interactivity involves multiple procedures. These are discussed in the remainder of this section.

3.5.1 Creating a Session

Next, developers must create their session handle. Depending on which graphics binding API is being used for the application, a call to the graphics binding's corresponding "`GetGraphicsRequirements`" function (specified by its extension) is required before calling the `xrCreateSession` function; this is needed in order to configure renderings and ensure that the rendered content appears on the developer's desired display [31]. Afterwards, a graphics binding structure can be established. These are also specified by the developer's graphics binding extensions and are chained onto the `next` pointer in `XrSessionCreateInfo`. This is one of the few times in the OpenXR Specification that the `next` pointer is used; structures in OpenXR contain a `type` field and a `void` pointer primarily for extension functionality [31]. Nevertheless, developers can then call `xrCreateSession` using their `XrSystemId` obtained from earlier.

```
XrResult xrCreateSession(
    XrInstance                                  instance,
    const XrSessionCreateInfo*                  createInfo,
    XrSession*                                  session);
typedef struct XrSessionCreateInfo {
    XrStructureType          type;
    const void*              next;
    XrSessionCreateFlags     createFlags;
    XrSystemId               systemId;
} XrSessionCreateInfo;
```

Figure 5 (Functions and Structures for Step 4 - Sessions)

3.5.2 Attaching Action Sets

Once a session has been established, developers must attach their `ActionSet`s to it so they are associated with the session. When doing this, the `Action`s and `ActionSet`s become immutable, meaning they no longer can be modified by the developer. Due to this, if the developer wishes to edit these `Action`s and `ActionSet`s in the future, the session must be destroyed. While this may seem like a pain, OpenXR purposely designed action setup to be done first in order to handle rebinding efficiently [31]. By providing users with the ability to instantly configure their interactions with a particular application via UIs, Monado can secretly map a developer's specified actions to the user's available hardware if no suggested bindings for their device exist [31]. This ultimately prevents application flows from being interrupted if rebinding were to occur more than once.

3.5.3 Building Reference and Action Spaces

If a developer wishes to add interactions with tracked objects, `XrSpace` handles are implemented in the application. These allow applications to directly create and specify frames of reference for tracking the physical world as well as determine how these frames of reference move relative to each other over a period of time [31]. There are two types of `XrSpace` handles: reference spaces and action spaces.

Reference spaces are those used to bootstrap an application's spatial reasoning; they are accessed with a developer's session and an `enum` [29]. While additional `enum`s can be introduced via extensions, the three main `enum`s in OpenXR are `STAGE` space, `LOCAL` space, and `VIEW` space. `STAGE` space is a bounded and standing play environment, while `LOCAL` space is a

bounded and seated play environment [31]. Both of these spaces are world-locked. However, `VIEW` space is a type of head space that is static or head-locked, making it beneficial for creating content that stays at a fixed point on HMDs [29, 31]. To obtain an `XrSpace` from any of these three `enums`, developers call the `xrCreateReferenceSpace` function.

Action spaces are used in an application to track `Pose` actions, such as a motion controller, over a period of time. To create them, developers call the `xrCreateActionSpace` function and provide their desired session and `Pose Action` [29]. It is important to note that this function creates an `XrSpace` similar to the `xrCreateReferenceSpace` function; both `XrSpace` handles have a particular session as the parent handle [31]. Additionally, for both `XrSpace` handle types, a fixed transform can be implemented at handle creation time. To find the transform from one space to another, developers can call `xrLocateSpace`. In OpenXR, spaces are always found with respect to another space.

```
C++
XrResult xrCreateReferenceSpace(
    XrSession                                   session,
    const XrReferenceSpaceCreateInfo*           createInfo,
    XrSpace*                                    space);
XrResult xrCreateActionSpace(
    XrSession                                   session,
    const XrActionSpaceCreateInfo*              createInfo,
    XrSpace*                                    space);

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

Figure 6 (Functions for Step 4 – Building Reference and Action Spaces)

3.5.4 Constructing Swapchains

As typical AR/VR applications will want to present rendered images to the user, OpenXR allows this by having the Monado runtime provide images that are organized via swapchains [29]. After obtaining the graphics API-specific formats through the `xrEnumerateSwapchainFormats` function, developers will then need to create one or more of these swapchains by calling `xrCreateSwapchain`. Once a swapchain is established, developers will access API-specific handles or references to the swapchain images by passing an array of extension-defined structures to the `xrEnumerateSwapchainImages` function [31]. The information received from this call is important for developers to save; it specifies where images are rendered in a graphics API-specific format that is used every frame [31].

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

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

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

Figure 7 (Functions for Step 4 - Swapchains)

3.6 Step 5: Participating in the Frame Loop

Within a frame loop for rendering, developers need to know the three functions that control the lifecycle of a frame. The first one, `xrWaitFrame`, is a scheduling call; the function blocks until the runtime determines when head-pose-dependent simulation and rendering is possible. The second one, `xrBeginFrame`, is a function executed by a developer's application in order to mark the start of rendering or GPU usage for a particular frame [31]. The third one, `xrEndFrame`, is a function that submits frames for display. While `xrBeginFrame` and `xrEndFrame` may be called from any thread, the developer's calls must be ordered as if they were single-threaded [31]. Moreover, developers must populate the `XrFrameEndInfo` structure's `displayTime` variable with the output obtained from `xrWaitFrame`; this is because `xrWaitFrame` notifies developers of when the next predicted display time is for future use in all calculations and space locations [31].

If the developer's application is using pipelined or multi-threaded rendering, OpenXR has more detailed timing requirements. For these renderings, at most one simultaneous `xrWaitFrame` call can be executed at a time. Each `xrWaitFrame` must also be eventually masked with a unique `xrBeginFrame`; this is because they come in pairs. For example, every `xrWaitFrame` has an `xrBeginFrame`, and every `xrBeginFrame` has an `xrWaitFrame` [31]. Moreover, any call to the `xrWaitFrame` function will be blocked in the Monado runtime until the previous frame's `xrBeginFrame` has been made [31].

```
XrResult xrWaitFrame(
    XrSession                                   session,
    const XrFrameWaitInfo*                      frameWaitInfo,
    XrFrameState*                               frameState);

XrResult xrBeginFrame(
    XrSession                                   session,
    const XrFrameBeginInfo*                     frameBeginInfo);

XrResult xrEndFrame(
    XrSession                                   session,
    const XrFrameEndInfo*                       frameEndInfo);

typedef struct XrFrameEndInfo {
    XrStructureType                               type;
    const void*                                   next;
    XrTime                                        displayTime;
    XrEnvironmentBlendMode                        environmentBlendMode;
    uint32_t                                      layerCount;
    const XrCompositionLayerBaseHeader* const*    layers;
} XrFrameEndInfo;
```

Figure 8 (Functions and Structures for Step 5 – Frame Loop)

Once it is time to render, developers must use the swapchain created earlier between `xrBeginFrame` and `xrEndFrame`. While the `xrAcquireSwapchainImage` function does not give a developer permission to write to an image, it does give the index of the swapchain in order to look up and create command buffers [31]. Typically after calling this function, the `xrWaitSwapchainImage` function is utilized before writing to an image [31]. However, as an optimization, developers can create their command buffers just by using the index from `xrAcquireSwapchainImage` to block on the compositor releasing the image for writing to an application [31]. When a developer is finished rendering, `xrReleaseSwapchainImage` is called before `xrEndFrame`; this is because `xrEndFrame` implicitly uses the most recently released swapchain image for displaying to a hardware device [31]. Throughout this rendering process in OpenXR, developers can find information about the predicted display time and pose that the head is anticipated to be in at a specific time by calling the `xrLocateViews` function. This function works similarly to `xrLocateSpace`.

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

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

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

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

Figure 9 (Functions for Step 5 - Swapchains)

3.6.1 Handling Input

After an application can successfully render, developers can focus on getting input in order to make their applications interactive. To do this, the `xrSyncActions` function should be called once every simulation frame; this function specifies the `ActionSet`s that should be active for a particular frame and updates all non-`Pose` input data in the active `ActionSet`s [31]. Once these `Action`s are synced, developers can obtain data through `xrGetActionState` calls. While these calls exist for each type of `Action` available, obtaining data from `Pose` actions is done somewhat differently; since tracking is latency and time sensitive, `Pose` actions constantly continue updating [31]. Due to this, developers can typically get data from a `Pose` action by creating its own `XrSpace` and calling `xrLocateSpace`. During this process, it is important to

note that only the active, inactive state of a `Pose` action is controlled by `xrSyncActions` [31]. Moreover, `ActionSet`s attached but not specified in the most recent call to the `xrSyncActions` function will cause the `Action`s in these sets to return as inactive. `Action`s may also not receive data if a session is not focused due to privacy and security concerns [31]. Nevertheless, developers can usually process a majority of their input either before calling `xrWaitFrame` and `xrBeginFrame`, or after calling `xrEndFrame` [31].

```
XrResult xrSyncActions(
    XrSession                                   session,
    const XrActionsSyncInfo*                    syncInfo);

XrResult xrGetActionStateBoolean(
    XrSession                                   session,
    const XrActionStateGetInfo*                 getInfo,
    XrActionStateBoolean*                       state);

XrResult xrGetActionStateFloat(
    XrSession                                   session,
    const XrActionStateGetInfo*                 getInfo,
    XrActionStateFloat*                         state);

XrResult xrGetActionStateVector2f(
    XrSession                                   session,
    const XrActionStateGetInfo*                 getInfo,
    XrActionStateVector2f*                      state);
```

Figure 10 (Functions for Step 5 - Input)

3.6.2 Polling Events

OpenXR maintains a per-instance event queue containing a variety of events, or messages sent from the runtime to the developer's application [29, 31]. Applications must poll from this queue regularly (typically once a simulation frame) [31]. To do this, developers can call the `xrPollEvent` function using a valid `XrInstance`. As many events occur only during a

particular session, these events can describe several changes, such as those related to a session state, an active interaction profile, or a continuity of reference spaces and tracking. Moreover, developers must provide a variable of type `XrEventDataBuffer` to `xrPollEvent` in order for the runtime to populate with an event of a different type. During this process, developers can reinterpret this structure accordingly by ensuring that the variable is of type `XrEventDataBuffer`; this is accomplished by checking its value after it returns from the call to `xrPollEvent`.

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

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

Figure 11 (Functions and Structures for Step 5 – Event Polling)

3.7 Summary of Steps

All in all, a typical OpenXR application with Monado compatibility follows a series of steps. First, developers create an instance by picking their desired extensions and API layers. Efforts are then redirected towards identifying the hardware system in use by the developer by finding an appropriate `XrSystemId` atom and `ViewConfigurationType enum`. Once an instance is established, developers focus on creating and customizing interactions in their application via `Action`s and `ActionSet`s. Afterwards, the developer's AR/VR `XrSession` handle is made; this includes attaching `ActionSet`s to the session handle, adding interactions

involving tracked objects through reference and action spaces, and building one or more swapchains. Then, the developer works with the frame loop to prepare image rendering by taking advantage of functions that control the lifecycle of a frame. Once an application is able to render, developers then concentrate on handling input data from actions and polling events to boost the application's interactivity. Ultimately, while the steps in this undergraduate thesis greatly simplify the AR/VR application-making process in OpenXR, they are intended to serve as a stepping stone for those interested in developing or learning more about standardized, immersive AR/VR experiences across a diverse set of hardware devices. As these platforms continue to improve, the OpenXR Specification will be an important and reliable resource for those wishing to obtain more in-depth information about how OpenXR and its many components operate.

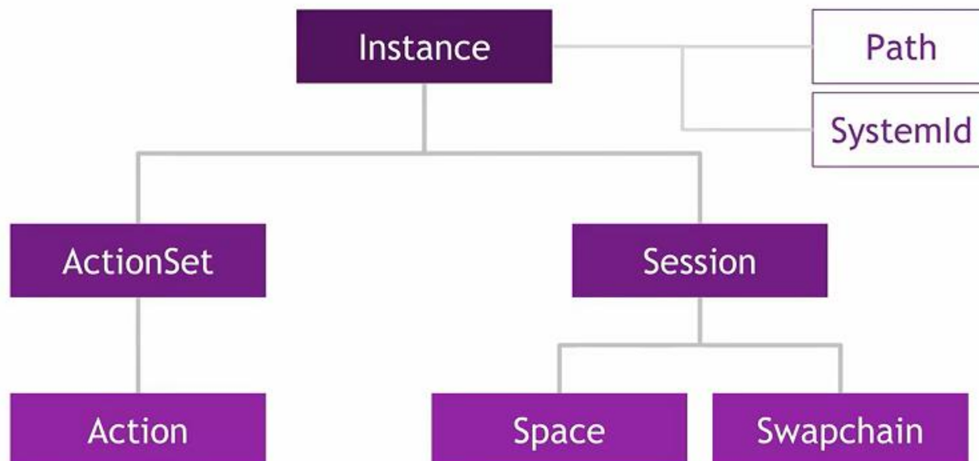

Figure 12 (OpenXR Hierarchy)

Chapter 4:
Conclusion

4.1 Key Takeaways

Ultimately, while OpenXR's standardizations are helping to reduce AR/VR fragmentation, developing AR/VR applications for any hardware system is still no easy feat. With a steep learning curve, complicated user interfaces, and current lack of in-depth resources to aid with debugging, programming for these platforms highlights how 2D concepts and designs cannot be applied to virtual 3D worlds. Due to this, AR/VR development is not only challenging for beginners with limited technical experience and general XR knowledge, but also a significant time investment for professionals designing immersive experiences. These roadblocks were very apparent while developing this thesis; originally starting as a Unity project aimed at developing real-world actions (such as closing a door or sitting in a chair) for the Quest 2, coding seamless C# applications required an extensive amount of math, geometry, and physics calculations. Even after creating a basic implementation, an exhausting amount of testing was still needed so the actions accurately reflected those seen in the physical world. Due to this, the project was greatly simplified by focusing on the movement of a 3D ball and its mechanical changes in Unity as it interacted with the user (via grabbing or throwing motions) or nearby objects (such as bouncing off walls or colliding with other balls) before turning into a project involving OpenXR and the Monado runtime. Nevertheless, the experience has taught me valuable lessons about this type of development. For example, there is still much we do not know about AR/VR development; with a gap in available resources, finding the answers to our programming questions requires a significant amount of personal backtracking, decision-making, and experimentation. Due to this,

optimizing for high performance rates is necessary in order to avoid motion sickness during testing. Mastering how to apply concepts from technical documentation as well as knowing how to download complex third-party libraries and SDKs (especially in a GNU/Linux environment) is also crucial to getting a project started and finding new approaches along the way. Ultimately, OpenXR and Monado can improve upon the development process by minimizing coding modifications and streamlining a developer's workflow. By preventing proprietary applications or SDKs from being created, developers can not only increase their productivity, but also promote the adoption of AR/VR technology to reduce confusion and fragmentation in the market. The work of this undergraduate thesis only scratches the surface; it can be expanded upon by including OpenXR examples of real-world actions and events that are supported by the Monado runtime. Additionally, the work can include more in-depth information about OpenXR's unique functions for elements such as instances, systems, spaces, view configurations, and sessions.

4.2 Future Work

The historic emergence of OpenXR and Monado has introduced new areas of research for AR/VR technologies. With the first proper release of OpenXR 1.0 and Monado 21.0.0 being less than three years old at the time of writing this thesis, implementers and developers are just beginning to obtain experience in developing extensions for the core API. In order to polish upcoming releases, future work can directed at providing support for Monado on other operating systems (such as Windows), adding face tracking to better integrate monoscopic fish tank views, and creating a dashboard that allows users to conveniently interact with an application's lifecycle

(start, pause, stop) and view the status of debug variables in real-time (such as frame statistics and timing) [28, 32]. Moreover, since OpenXR and Monado are complex platforms aiming to be the standard for AR/VR developers of all skill levels, efforts can be directed at making more educational content, such as online resources, tutorials, videos, or coding samples, that thoroughly explain the basics behind programming AR/VR OpenXR-compliant applications in order to be more approachable to those unfamiliar with the standard. Ultimately, as the exciting work on OpenXR and Monado continues to gain substantial momentum in the XR industry and ecosystem, software developers and hardware vendors alike will shape the vibrant, cross-platform standard not only to ensure that enterprise communities have easy-access to the best XR technology available today, but also to deliver innovative and next-generation AR/VR experiences across a broad range of platforms and devices.

BIBLIOGRAPHY

[1] Tulane University: School of Professional Advancement, “What’s the Difference Between AR and VR?” *Tulane University: School of Professional Advancement.* [Online]. Available: https://sopa.tulane.edu/blog/whats-difference-between-ar-and-vr. [Accessed March 31, 2022].

[2] M. E. Porter and J. E. Heppelmann, “Why Every Organization Needs an Augmented Reality Strategy,” *Harvard Business Review*, December, 2017. [Online]. Available: https://hbr.org/2017/11/why-every-organization-needs-an-augmented-reality-strategy. [Accessed April 2, 2022].

[3] Aniwaa, “Types of VR headsets: PC VR, standalone VR, smartphone VR,” *Aniwaa*, August 6, 2021. [Online]. Available: https://www.aniwaa.com/guide/vr-ar/types-of-vr-headsets/. [Accessed March 30, 2022].

[4] Meta Quest, “Quest 2,” *Facebook Technologies*, 2022. [Online]. Available: https://www.oculus.com/quest-2/. [Accessed March 30, 2022].

[5] Oculus, “Get Started Developing for the Oculus Quest Platform,” *Facebook Technologies*, 2022. [Online]. Available: https://developer.oculus.com/quest/. [Accessed March 28, 2022].

[6] Oculus VR, “Introducing Oculus Quest 2, the Next Generation of All-in-One VR,” *Facebook Technologies*, September 16, 2020. [Online]. Available: https://developer.oculus.com/blog/introducing-oculus-quest-2-the-next-generation-of-all-in-one-vr/. [Accessed March 29, 2022].

[7] M. Clark, “Oculus’ Passthrough API will enable experiences that mix VR and the real world,” *The Verge*, July 23, 2021. [Online]. Available: https://www.theverge.com/2021/7/23/22590794/oculus-mixed-reality-api-quest-2-development-kit-unity. [Accessed March 28, 2022].

[8] M. Vovk, “Microsoft HoloLens 2 - Mixed Reality headset designed to get work done,” *BE terna*, February 11, 2021. [Online]. Available: https://www.be-terna.com/insights/microsoft-hololens-2-mixed-reality-headset-designed-to-get-work-done. [Accessed March 30, 2022].

[9] Microsoft, “HoloLens 2,” *Microsoft*, 2022. [Online]. Available: https://www.microsoft.com/en-us/hololens/hardware. [Accessed April 1, 2022].

[10] Microsoft, “Start Developing for Mixed Reality,” *Microsoft*, 2022. [Online]. Available: https://www.microsoft.com/en-us/hololens/developers. [Accessed April 1, 2022].

[11] Microsoft, “What is the Mixed Reality Toolkit,” *Microsoft*, November 29, 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows/mixed-reality/mrtk-unity/?view=mrtkunity-2021-05. [Accessed April 2, 2022].

[12] Google, “Choose Your Development Environment,” *Google*, May 20, 2020. [Online]. Available: https://developers.google.com/cardboard/develop. [Accessed March 28, 2022].

[13] J. Chen, “Open sourcing Google Cardboard,” *Google Developers*, November 6, 2019. [Online]. Available: https://developers.googleblog.com/2019/11/open-sourcing-google-cardboard.html. [Accessed March 28, 2022].

[14] J. Vincent, “Google's new VR View tool allows easy embedding of 360-degree content,” *The Verge*, March 31, 2016. [Online]. Available: https://www.theverge.com/2016/3/31/11336386/google-vr-view-360-degree-content-embed. [Accessed March 31, 2022].

[15] Google, “Create immersive VR experiences,” *Google Developers*, 2022. [Online]. Available: https://developers.google.com/cardboard. [Accessed April 1, 2022].

[16] J. Valcarcel, “Google Cardboard Is VR's Gateway Drug,” *Wired*, May 28, 2015. [Online]. Available: https://www.wired.com/2015/05/try-google-cardboard/. [Accessed March 28, 2022].

[17] Google, “Google Cardboard,” *Google VR*, 2022. [Online]. Available: https://arvr.google.com/cardboard/. [Accessed March 31, 2022].

[18] Unity, “Augmented Reality and Virtual Reality Games,” *Unity Technologies*, 2022. [Online]. Available: https://unity.com/solutions/ar-and-vr-games. [Accessed April 1, 2022].

[19] Epic Games, “Unreal Engine for extended reality (XR),” *Epic Games*, 2022. [Online]. Available: https://www.unrealengine.com/en-US/xr. [Accessed March 31, 2022].

[20] Epic Games, “Blueprint Visual Scripting,” *Epic Games*, 2022. [Online]. Available: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/. [Accessed March 29, 2022].

[21] PTC, “Vuforia: Market-Leading Enterprise AR,” *PTC*, 2022. [Online]. Available: https://www.ptc.com/en/products/vuforia. [Accessed April 2, 2022].

[22] Circuit Stream, “Ultimate AR Comparison Guide,” *Circuit Stream*, September 3, 2021. [Online]. Available: https://circuitstream.com/blog/augmented-reality-guide/. [Accessed April 2, 2022].

[23] Wikitude, “Wikitude Augmented Reality: the World's Leading Cross-Platform AR SDK,” *Wikitude*, 2022. [Online]. Available: https://www.wikitude.com/. [Accessed March 30, 2022].

[24] The Khronos Group, “OpenXR,” *The Khronos Group*, 2022. [Online]. Available: https://www.khronos.org/api/index_2017/openxr. [Accessed April 1, 2022].

[25] LunarG, “OpenXR Services,” *LunarG*, 2022. [Online]. Available: https://www.lunarg.com/services/openxr-services/. [Accessed April 1, 2022].

[26] J. Bornecrantz, “Introducing: Monado,” *Collabora*, March 18, 2019. [Online]. Available: https://www.collabora.com/news-and-blog/news-and-events/introducing-monado.html. [Accessed March 31, 2022].

[27] Monado, “Monado - Open Source XR Platform,” *Collabora*, 2022. [Online]. Available: https://monado.dev/. [Accessed April 1, 2022].

[28] Monado, “Monado - XR Runtime (XRT),” *Collabora*, 2022. [Online]. Available: https://monado.freedesktop.org/. [Accessed March 31, 2022].

[29] The Khronos Group, “The OpenXR Specification,” *The Khronos Group*, 2022. [Online]. Available: https://www.khronos.org/registry/OpenXR/specs/1.0/html/xrspec.html. [Accessed April 1, 2022].

[30] The Khronos Group, “OpenXR 1.0 Reference Guide,” *The Khronos Group*, 2021. [Online]. Available: https://www.khronos.org/files/openxr-10-reference-guide.pdf. [Accessed April 1, 2022].

[31] R. Pavlik, “Unifying Reality: Building Experiences with OpenXR | Laval Virtual 2020,” *YouTube*, June 5, 2020. [Video file]. Available: https://www.youtube.com/watch?v=F6jZCwko1Qs. [Accessed April 1, 2022].

[32] Monado, “Monado - GSoC ideas 2022,” *Collabora*, 2022. [Online]. Available: https://monado.freedesktop.org/gsoc-2022.html. [Accessed April 2, 2022].

ACADEMIC VITA

ANASTASIA SEREMULA

ams9248@psu.edu

https://www.linkedin.com/in/anastasiaseremula

EDUCATION

The Pennsylvania State University, University Park, PA — May 2022

Bachelor of Science in Computer Science (Schreyer Honors College)

Dean's List: Fall 2018 - Spring 2022

SKILLS

- **Languages:** Java, C++, C, MySQL, PHP, MATLAB, Verilog, HTML, CSS
- **Operating Systems:** Windows, UNIX, Linux
- **Software:** BlueJ, Vivado, Unity, WordPress, Microsoft Office, Office 365, MS Visual Studio, MS Publisher, Adobe Photoshop, Adobe Illustrator, Adobe InDesign

WORK EXPERIENCE

Penn State Lehigh Valley, Center Valley, PA — August 2019 - May 2022

Peer Tutor

- Provided and assisted in integrating effective learning and study strategies for students in various math, computer science, and IST courses.
- Conducted individual and group peer tutoring sessions on a weekly basis to reinforce course content, assignments, and materials.
- Prioritized and developed strong relationships with tutees in order to better evaluate their academic and socio-emotional needs.

PROGRAMMING PROJECTS

Database Design Project (SQL) — December 2021

Lead Developer

- Designed, developed, and implemented a database intended to act as a personal organizational guide for a popular mobile game. Created an ER-model and translated this model to relational schema to build a management system that stores, manipulates, and searches for data. Written in MySQL, PHP, and HTML.

Dynamic Memory Allocation Functions (Operating Systems) — October 2021

Lead Developer

- Developed a dynamic storage allocator with a partner for incorporating a customized version of the malloc, free, and realloc functions. Designed a segregated free list for finding memory as well as managed memory on the heap while optimizing availability with coalescing to avoid fragmentation. Written in C.

Networking with Transmission Control Protocol (TCP) — April 2020

Lead Developer

- Designed and developed a client and server application with a group of students to copy a file on the client computer to a file on the server computer and vice versa, mimicking the UNIX/Linux remote copy command that allows users to copy files from and to remote machines. The applications are written in C and also have manual pages.

Hangman Application November 2019
Lead Developer

- Collaborated with a team to design, develop, and implement an application to play Hangman, a guessing game where the user tries to figure out the computer's secret word by suggesting letters within a certain number of guesses. The application integrates a graphical user interface, database, and object-oriented programming using Java.

CLUBS AND ORGANIZATIONS

Council of Commonwealth Student Governments (CCSG) May 2020 – August 2020
Executive Director of Public Relations

- Directed CCSG's media relations activities, including the production and dissemination of press releases, media advisories, and public announcements.
- Created and maintained content for internal and external communications, serving as a frontline point of contact for the organization.
- Worked with internal departments, including marketing, IT, and social media, to utilize data analytics and generate content for CCSG's website/social media that will increase online visibility and traffic.
- Built relationships with organization partners, internal teams, external media and commonwealth representatives to help achieve CCSG's initiatives and objectives.

PSU-LV Student Government Association 2019-20 April 2019 – May 2020
Vice-President

- Instrumental in doubling SGA membership and developing training strategies to build required skill sets in Senators and Executive Board members for successful adoption of new solutions.
- Served as the speaker of the Student Senate, which is composed of student leaders that pass legislation and address student concerns affecting more than 1000 students on campus.
- Developed relationships with campus faculty, staff, students, and clubs to successfully integrate new events and initiatives.
- Managed the implementation of SGA programs and activities: Major in a Minute, Financial Literacy, SGA Town Hall, etc.

PSU-LV Advisory Board April 2019 – May 2020
Student Representative

- Served as a community liaison supporting the Penn State Lehigh Valley campus with a group of alumni, business professionals, and community individuals.
- Provided input into campus decisions and helped identify potential donors that might support PSU-LV students and programs.
- Promoted University programs and connected with area legislators in supporting university initiatives.
- Represented Penn State Lehigh Valley in the community by being knowledgeable about programs, initiatives, and opportunities to give time and financial support to the campus.

PSU-LV Student Fee Board April 2019 – May 2020
Student Representative

- Attended biweekly meetings with appointed students and faculty members to review proposals based on identified student needs.

- Voted on proposals that are focused on improving curricular opportunities, providing specialized services and programs for diverse populations, enriching the campus environment, or encouraging the involvement of students.
- Recommended stipulations of the applicant(s) in order to allocate activity and facility fund monies consistent with the purpose of the Student Fee collected with tuition.

PSU-LV Student Government Association 2018-19 September 2018 – April 2019
Secretary

- Recorded meeting minutes and distributed to Student Senators through Canvas in order to document important decisions and ideas.
- Managed weekly attendance and updated absent Student Senators on important information missed in previous meetings.
- Announced upcoming meetings and events to organization to increase Senator participation.
- Communicated with President and Vice President to propose future event dates and activities.

HONORS/AWARDS

President Freshman Award 2019
High Cumulative Average: College of Engineering 2019
English Department Award for Exceptional Achievement in Expository Writing 2019
Penn State Lehigh Valley Student Art Exhibition 2019